# AN IMPLEMENTATION OF THE IEEE 1609.4 WAVE STANDARD FOR USE IN A VEHICULAR NETWORKING TESTBED

by

Kyle Kuffermann

A Thesis Submitted to the Faculty of
The College of Engineering & Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Science

Florida Atlantic University
Boca Raton, FL
December 2014

**AN IMPLEMENTATION OF THE IEEE 1609.4 WAVE STANDARD FOR USE IN A VEHICULAR NETWORKING TESTBED**

by

Kyle Kuffermann

This thesis was prepared under the direction of the candidate's thesis advisor, Dr. Imad Mahgoub, Department of Computer & Electrical Engineering and Computer Science, and has been approved by the members of his supervisory committee. It was submitted to the faculty of the College of Engineering & Computer Science and was accepted in partial fulfillment of the requirements for the degree of Master of Science.

SUPERVISORY COMMITTEE:

[illegible]
Imad Mahgoub, Ph.D.
Thesis Advisor

[illegible]
Mohammad Ilyas, Ph.D.

[illegible]
Shihong Huang, Ph.D.

[illegible]
Nurgun Erdol. Ph.D.
Chair, Department of Computer & Electrical Engineering and Computer Science

[illegible]
Mohammad Ilyas, Ph.D.
Dean, The College of Engineering & Computer Science

[illegible]
Deborah L. Floyd, Ed.D.
Interim Dean, Graduate College

11/18/2014
Date

## ACKNOWLEDGEMENTS

I would like to express my gratitude and appreciation to my advisor, Dr. Imad Mahgoub, for encouraging my research and helping me to grow as a computer engineer. His guidance and positivity proved to be invaluable. I would also like to thank my committee members, Dr. Mohammad Ilyas and Dr. Shihong Huang, for taking the time to review my work and for providing beneficial feedback. Thank you to the members of FAU's Smart Mobile Computing research group for their input and support, particulary Monika for her helpful suggestions and for taking the time to proofread my thesis; and Lina for her help with creating diagrams. I would like to thank Todd, Jose, Mike, and Doug from tkLABS for offering suggestions, answering questions, and the outstanding work performed in support of this implementation. Thank you to my parents and family for their love and support; they have always encouraged me to work hard in order to acheive my goals.

Lastly, I would like to dedicate this work to my wife, Kelly, who was a constant source of inspiration and motivation, without whom I would have not been able to accomplish this work.

This work is supported by the National Science Foundation under MRI grant #1229616.

## ABSTRACT

| | |
|---|---|
| Author: | Kyle Kuffermann |
| Title: | An Implementation of the IEEE 1609.4 WAVE Standard for Use in a Vehicular Networking Testbed |
| Institution: | Florida Atlantic University |
| Thesis Advisor: | Dr. Imad Mahgoub |
| Degree: | Master of Science |
| Year: | 2014 |

We present an implementation of the IEEE WAVE (Wireless Access in Vehicular Environments) 1609.4 standard, Multichannel Operation. This implementation provides concurrent access to a control channel and one or more service channels, enabling vehicles to communicate among each other on multiple service channels while still being able to receive urgent and control information on the control channel. Also included is functionality that provides over-the-air timing synchronization, allowing participation in alternating channel access in the absence of a reliable time source. Our implementation runs on embedded Linux and is built on top of IEEE 802.11p, as well as a customized device driver. This implementation will serve as a key component in our IEEE 1609-compliant Vehicular Multi-technology Communication Device (VMCD) that is being developed for a VANET testbed under the Smart Drive initiative, supported by the National Science Foundation.

**AN IMPLEMENTATION OF THE IEEE 1609.4 WAVE STANDARD FOR USE IN A VEHICULAR NETWORKING TESTBED**

## LIST OF FIGURES

# CHAPTER 1
# INTRODUCTION

There are many applications in which vehicular networks can be used to improve both the safety and comfort of drivers. Among safety applications are collision avoidance, navigation assistance, and alerts that notify the driver of speed limits, work zones, or road obstacles [23]. Non-safety applications include providing weather and traffic information, and locating Points of Interest [23], as well as infotainment applications such as Internet access, games, and multimedia applications [24].

In order to provide these benefits to drivers, vehicles must be equipped with communications devices that conform to a standardized technology in order to ensure that all vehicles can be part of the network, regardless of manufacturer. Because of this, the WAVE (Wireless Access in Vehicular Environments) set of standards was developed by IEEE. According to [8]:

> WAVE provides a communication protocol stack optimized for the vehicular environment, employing both customized and general-purpose elements.

This work presents an implementation of the IEEE 1609.4 standard [8] that will function as part of a full WAVE protocol stack running on our Vehicular Multi-technology Communication Device (VMCD) [25]. This device will be used in a vehicular networking testbed as part of the Smart Drive initiative supported by the National Science Foundation. To be included on this device are all of the hardware and software components for the development and evaluation of vehicular networking protocols and applications. The IEEE 1609.4 implementation described here is a key component of this system.

## 1.1 VANET

The vehicular networking described in this work is referred to as Vehicular Ad-hoc Networking (VANET), which is a subset of Mobile Ad-hoc Networking (MANET) within the field of wireless ad-hoc networks. A wireless ad-hoc network is one in which the communicating nodes do not rely on a centralized infrastructure. Rather than using routers, switches, and access points, as with a standard 802.11 wireless network, all nodes in a wireless ad-hoc network act as routers as well as being the end-users [26]. Because of this, message transmission among nodes in an ad-hoc network is typically done in multi-hop fashion, where intermediate nodes relay data from a source node to the destination node [26].

Wireless ad-hoc networks are used in mobile environments due to the highly dynamic nature of these environments. As the network size and density grows and shrinks, there is no need to make adjustments to an infrastructure to support these changes. In MANETs, nodes may move in any direction, adding to the complexities of maintaining routes among nodes.

VANETs are a special case of MANETs in which the nodes are vehicles. There are some characteristics that are unique to VANETs, such as: mobility is constrained to roads, nodes move at a high rate of speed, and power consumption is not a primary concern. As mentioned in the previous chapter, VANETs can be used in order to provide both safety and non-safety services to drivers. Widespread use of VANETs allows the possibility of saving lives, as well as improving the overall experience of driving.

The development of the IEEE WAVE standards is one step in the direction of mainstream VANET use, as it aims to standardize the technologies used. According to [27], U.S. regulators have begun working on a mandate for early 2017 to require this vehicle-to-vehicle communication technology to be included on all new vehicles. According to ITS (Intelligent Transportation Systems) America, President and CEO

Scott Belcher [28]:

> Motor vehicle crashes are the leading cause of death for children and young adults in the United States, with an annual death toll of 33,000, over 2.3 million injuries, and an economic cost of $871 billion. ...[vehicular communication] technology could potentially prevent or reduce the impact of 80 percent of all unimpaired vehicle crashes, saving thousands of lives each year and dramatically reducing the nearly $1 trillion cost of crashes and congestion to American families and our nations economy

Implementing and evaluating these standards is the next step towards implementing VANETs and receiving these benefits.

## 1.2 PROBLEM STATEMENT

With the use of vehicular networks for communication of both safety and non-safety information, a significant concern is that traffic from non-safety information will interfere with and disrupt important safety information [23]. A solution for this is to keep safety and non-safety traffic on separate channels. Although this does solve the problem of non-safety traffic interfering with safety traffic, it creates another problem: each device would require two physical radios in order to participate in both safety and non-safety applications simultaneously. This can add significant expenses to the manufacturing of vehicular networking devices.

The 1609.4 standard addresses this issue by specifying functionality that allows for a device with a single radio to communicate on multiple channels concurrently. This allows a device to participate in non-safety applications, while still being able to receive urgent safety information without the need for a second physical radio. In order to acheive this, the 1609.4 standard specifies channel switching at regular intervals, as well as management functions that support this channel switching. A detailed description of the functionality specified by this standard is included in Chapter 3.

As our device will be used for researching vehicular networking protocols and applications, it is important that we support functionality specified by the IEEE standards. Much of the research and experiments in the area of vehicular networking have been done using other existing communications standards [1]. These results may vary from real-world scenarios due to differences in the underlying technologies. There have been some existing implementations of the 1609.4 standard, but most have been done using simulations, are incomplete, or are based on older, trial-use versions of the standard. These existing implementations are described in section 1.3. Our goal here was to create an implementation of this standard that fills in the gaps left by previous implementations, and is compatible with the hardware on our VMCD.

## 1.3 RELATED WORK

Since the new version of the IEEE 1609.4 standard was ratified in 2010, there have been some implementations using simulators, such as ns-2 and Qualnet [2][3][6]. There have also been hardware-based implementations based on the previous draft versions of the 1609.4 standard [4][5]. There have been relatively few implementations of the current 1609.4 standard that run on hardware and can be used in real-world scenarios, and the ones that do are incomplete. [1], for example, makes no mention of implementation of Timing Advertisement frames or Vendor Specific Action frames. The rest of this section describes implementations based on the *mac80211*, *cfg80211*, and *ath5k* Linux kernel modules [1] [11], an implementation focused on time synchronization [12], and an implementation that provides access to multiple service channels concurrently [2].

[1] presents an IEEE 802.11p [9] and 1609.4 [8] implementation that runs on a Linux platform, by modifying the existing *ath5k* driver and the *mac80211* and *cfg80211* kernel modules. The *ath5k* module is an open-source driver for the wireless chipset used on their hardware platform. The *mac80211* module implements the MAC (Medium Access Control) sublayer for the IEEE 802.11 standard. The

*cfg80211* module is used for configuration of an 802.11 wireless device. They have created extensions to these existing modules in order to implement IEEE 802.11p and IEEE 1609.4. Their implementation does include the *OCBEnabled* operation of 802.11p, as well as the core channel-switching functionality of 1609.4. In this implementation, synchronization to UTC time is acheived through the use of GPS (Global Positioning System). They do not, however, include any functionality to synchronize to UTC (Universal Coordinated Time) in the event that a GPS signal is not available. They also have not explored the possibility that a device may be participating in multiple services on multiple service channels concurrently. This paper also includes performance data showing the average delay on both the control channel (CCH) and service channel (SCH) as the bitrate is increased on the SCH, as well as the average delay for both low and high-priority traffic as the low-priority traffic is increased beyond the maximum channel capacity. This data shows that the control channel traffic is independent of the service channel traffic, and that the EDCA (Enhanced Distributed Channel Access) priority mechanism is working as intended, with there being considerably less delay for high-priority messages.

[11] also contains an implementation based on the *ath5k*, *mac80211*, and *cfg80211* modules. This implementation, although published more recently, references the old trial-use versions of the WAVE standards. In their implementation, multi-channel operation occurs within a WAVE Basic Service Set (WBSS). In the current version of the WAVE standards, there is no notion of a WBSS [8][10][7].

[12] presents an implementation that focuses on time synchronization using the Timing Advertisement mechanism specified in the IEEE WAVE standards. This paper contains a notable modification to the standard in the transmission of Timing Advertisements (TAs). In the 1609.4 standard, the MLME (MAC sublayer Management Entity) populates the timestamp to be sent in the TA frame. However, there is some amount of time that passes between the time that the timestamp is populated

and when it is actually transmitted, causing inaccuracy of the timestamp value sent over the air. In order to minimize this inaccuracy, they populate the timestamp value in the lower MAC instead of in the MLME. This paper does not describe in detail the other features of 1609.4 implemented, nor the platform or simulator that it was implemented on.

[2] contains an implementation of the 1609.4 standard using the ns-2 simulator. This paper focuses on the channel switching portion of the standard, and proposes a few enhancements. This paper brings up an interesting point about some ambiguities in the standard:

> ... other details were also not defined in the current 1609.4 standard, or left open to user-implementations, **such as the channel selector policy to adopt to decide on the channel to be served at each SCH interval when the vehicle is involved with multiple services running at different channels**.

This introduces the concept of multiple concurrent service channel access, something that is not explicitly stated in the standard, but is implied, due to the availability of 6 service channels in WAVE. This is the only known implementation of concurrent multiple SCH access. They have implemented 2 different policies for channel scheduling: round robin, and earliest deadline first. As far as we are aware, there are no existing implementations of this running on a physical device that can be used for testing in real-world, on-road scenarios.

## 1.4 CONTRIBUTIONS

The contributions of this work are as follows:

- Implementation of the IEEE 1609.4 standard, including:
  - Support for transmission of both WSMP (WAVE Short Message Protocol) and IPv6 data. This allows applications to send messages using the WSMP

protocol specified in IEEE 1609.3 [10], on either the control channel or a service channel, and to send messages on a service channel using the existing IPv6 protocol.

  - Support for all specified channel access modes: *continuous*, *alternating*, *immediate*, and *extended*. This allows a device to participate in one or more communication services concurrently without requiring multiple physical radios.
  - Support for multiple concurrent service channel access. This is an extension to the alternating channel access mode, where alternating access is provided on multiple service channels in addition to the control channel.
  - Support for timing synchronization, including functionality to set and get the device's UTC time estimate, as well as the ability to transmit and receive timing information over-the-air using Timing Advertisements. Timing Advertisements are used to provide synchronization to a device that does not have a reliable source of UTC time.
  - Management of configuration and status information required by the standard.

- Testing and Verification of the 1609.4 implementation, including:
  - Verifying that application data is transmitted on the correct channel for data identified as either WSMP or IPv6.
  - Verifying that channel switches occur at the right times for all channel access modes and combinations of access modes.
  - Verifying that the system's UTC time estimate is updated correctly, and that Timing Advertisements are transmitted according to the repeat rate specified.

## 1.5 ORGANIZATION

Following this chapter, the organization of this thesis is as follows: Chapter 2 describes the WAVE set of standards from a high-level, as well as providing an in-depth description of IEEE 1609.4. Chapter 3 describes our implementation, including the platform and development environment, our approach, and the implementation of the primitives specified by the standard. Chapter 4 describes testing of our implementation, including tools used, testing procedures, and the results of these tests. Chapter 6 describes our conclusions as well as future research opportunities.

# CHAPTER 2
# IEEE WAVE STANDARDS

This chapter provides an overview of the protocol stack specified in the IEEE WAVE standards [7] [8] [10], followed by a more in-depth description of the functionality specified in IEEE 1609.4 [8].

## 2.1 WAVE ARCHITECTURE

WAVE operation is described in terms of *services* and specified in terms of *primitives*. The services are divided into data plane services and management plane services. The data plane consists of the communication protocols in order to transmit data between devices. The management plane consists of functionality that supports the data plane, but does not pass application data directly. The architecture of the WAVE protocol stack is shown in Figure 2.1. The standards correspond to the layers of the stack as follows:

**802.11p** This is shown at the bottom of 2.1, and includes the Physical layer (PHY), Physical Layer Management Entity (PLME), MAC sublayer Management Entity (MLME), and the lower half of the WAVE MAC.

**1609.4** This includes the upper half of the WAVE MAC as well as the MLME Extension.

**1609.3** This includes the rest of the data plane shown, including the Logical Link Layer (LLC), WAVE Short Message Protocol (WSMP), IPv6, and UDP/TCP protocols, as well as the WAVE Management Entity.

**1609.2** This includes the WAVE Security Services portion of the Management Plane.

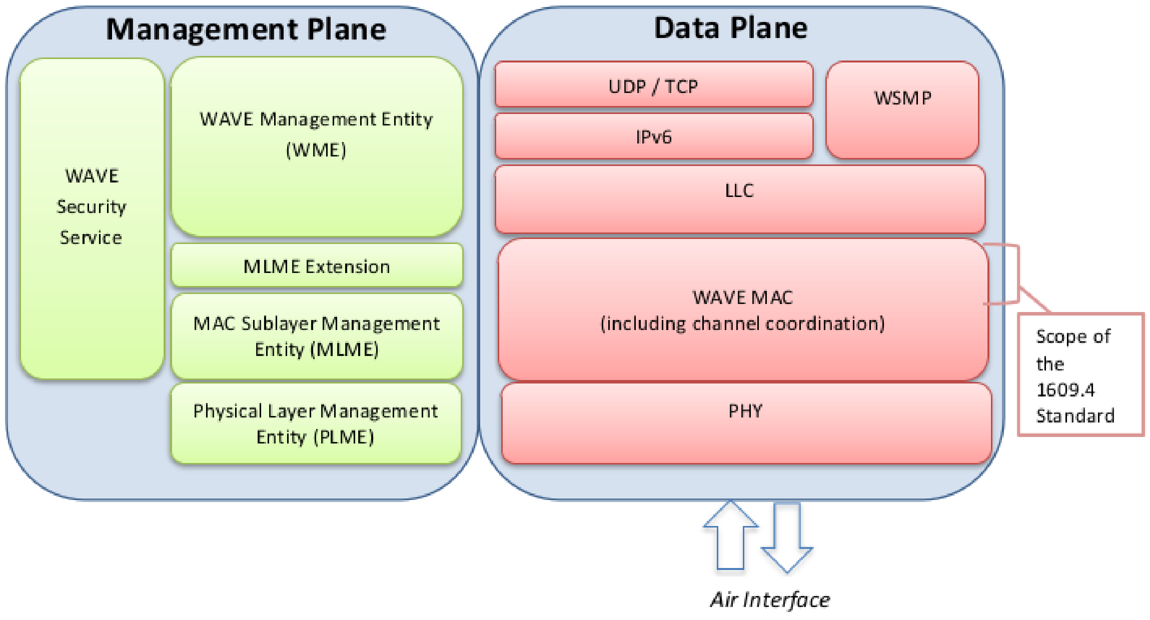

Management Plane
WAVE Security Service
WAVE Management Entity (WME)
MLME Extension
MAC Sublayer Management Entity (MLME)
Physical Layer Management Entity (PLME)
Data Plane
UDP / TCP
IPv6
WSMP
LLC
WAVE MAC (including channel coordination)
PHY
Scope of the 1609.4 Standard
Air Interface

**Figure 2.1**: Architecture of the WAVE protocol stack [8]

IEEE 802.11p provides the foundation for the WAVE set of standards. 802.11p is an ammendment to the 802.11 standard for use in vehicular networks. This specifies the functionality for the physical layer, as well as the MAC sublayer. The core feature of 802.11p is allowing communication between devices without the need to join a basic service set (BSS). This is referred to in the standard as OCBEnabled operation, or Outside the Context of a BSS operation. The reason for this operation is because of the amount of time that it takes 802.11 devices to set up a connection using scanning and association. Since vehicles are constantly moving at a high rate of speed, there is a limited amount of time for communication, and it does not make sense to use a large fraction of this limited communication time for association.

Above 802.11p are the 1609 set of standards. 1609.4 specifies an extension to the MAC sublayer allowing for multi-channel communications. This allows On-Board Units (OBUs) or Road-Side Units (RSUs) that only have one radio to concurrently communicate on multiple channels. 1609.4 specifies functionality for channel switching and timing synchronization. This includes routing data received from the LLC to the proper channel, using 802.11's EDCA functionality per channel, regulating channel access, sending Vendor Specific Action (VSA) and Timing Advertisement (TA) frames, and MAC address changes for pseudonymity.

The 1609.3 standard specifies the network and transport layer services. This includes WAVE Service Advertisements, the WAVE Short Message Protocol (WSMP), and how the existing LLC and IPv6 protocols are to be used in a WAVE system. WAVE Service Advertisements are used to announce the availability of a service, and are transmitted by a service provider. The WAVE Short Message Protocol is used for low-latency transmission of messages, allowing the applications to set physical layer transmission parameters per message.

A WAVE device is defined as a device that conforms to the 802.11p, 1609.4, and 1609.3 standards [7]. However, there are additional standards currently available, as well as ones still in development. 1609.2 specifies security services, 1609.5 Communication Manager and 1609.6 Remote Management Services are currently open / in development, 1609.11 is an application-level standard that specifies an over-the-air payment protocol, and 1609.12 specifies identifier allocations.

## 2.2 IEEE 1609.4: MULTICHANNEL OPERATION

As mentioned in the previous section, operation of the WAVE standards is specified using primitives. These are analogous to function calls, and specify interaction between different layers or entities in the WAVE standards. They are exchanged across Service Access Points (SAPs), which are defined interfaces between different entities in the standard. The 1609.4 standard specifies primitives for the MAC SAP and the MLMEX SAP. The MAC SAP defines the interface between the MAC sublayer and the LLC sublayer. The MLMEX SAP defines the inerface between the WAVE

Management Entity (WME), specified in 1609.3, and the MAC sublayer management extension. These SAPs are shown in Figure 2.2.

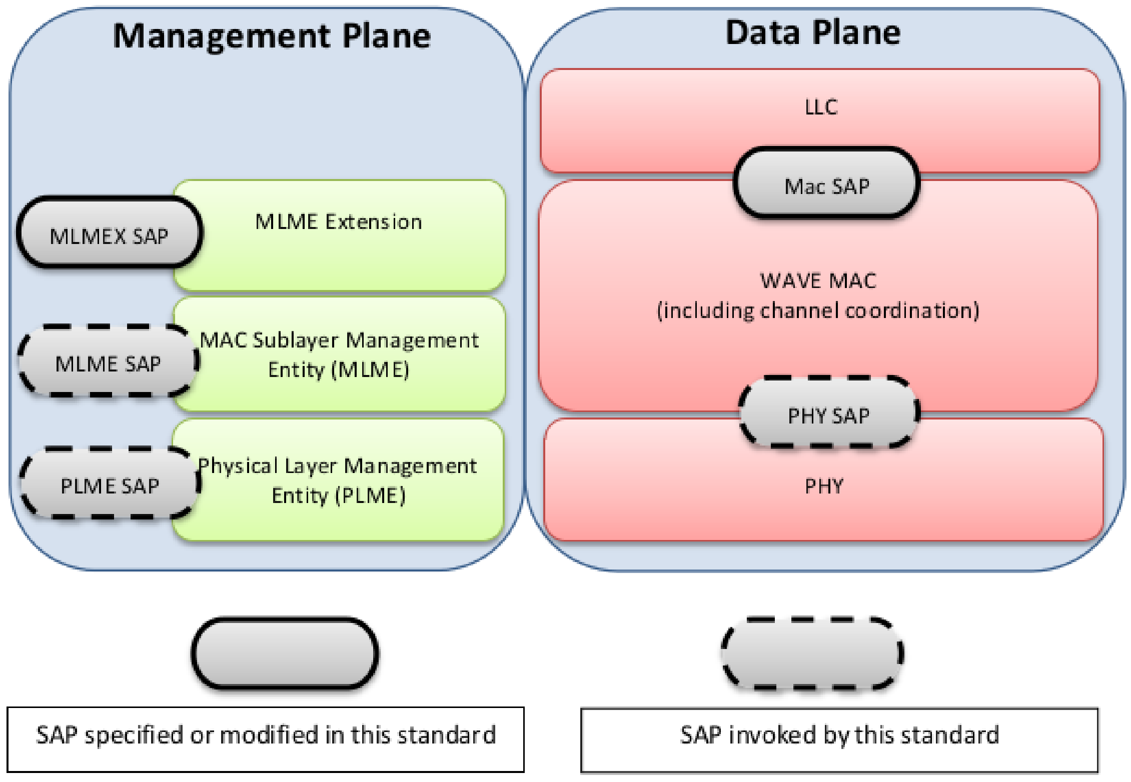

Management Plane
Data Plane
LLC
Mac SAP
WAVE MAC
(including channel coordination)
PHY SAP
PHY
MLMEX SAP
MLME Extension
MLME SAP
MAC Sublayer Management Entity (MLME)
PLME SAP
Physical Layer Management Entity (PLME)
SAP specified or modified in this standard
SAP invoked by this standard

**Figure 2.2**: Service Access Points used in 1609.4 [8]

In order to implement the primitives specified (or modified) at the MAC SAP and MLMEX SAP, the 1609.4 entities invoke primitives at the PHY, MLME, and PLME SAPs. The PHY SAP is the interface between the WAVE MAC sublayer and the physical layer, the MLME SAP is the interface to the 802.11p MAC sublayer management entity, and the PLME is the interface to the PHY sublayer management entity. The primitives are used to specify the desired operation, but the implementation of the primitives is not defined in the standard. Implementation details are left to the implementer of the standard.

### 2.2.1 Transmit Operation

1609.4 describes the transmit operation using a reference architecture; actual implementations may vary. This reference architecture illustrates the transmit operation on 2 channels: a control channel (CCH), and a service channel (SCH). It is shown in Figure 2.3. For each channel, there exists a separate instance of the 802.11 MAC sublayer, with its own set of EDCA queues. EDCA (Enhanced Distributed Channel Access) is a priority access mechanism specified in 802.11 (first introduced in 802.11e, 2005). The rest of this section will describe the EDCA mechanism and the channel routing functionality that make up the transmit side operation of the WAVE MAC.

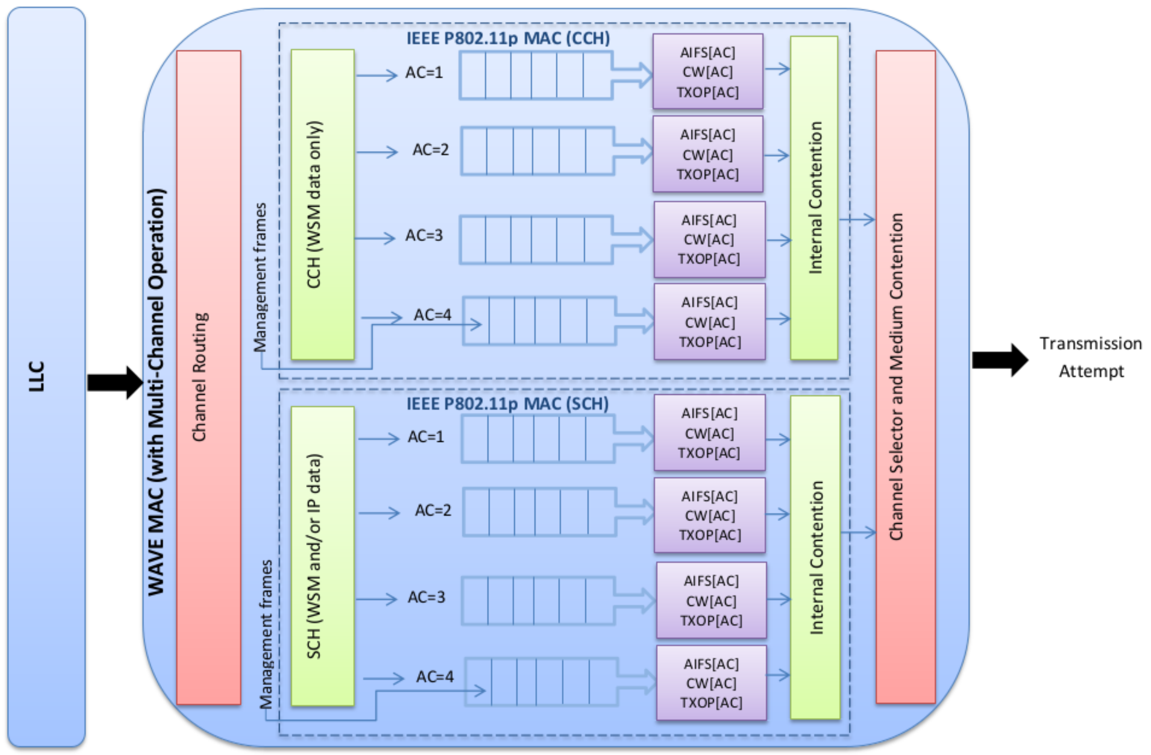


**Figure 2.3**: Architecture of 1609.4's Transmit Operation [8]

*EDCA Mechanism*

In the EDCA mechanism (for a single channel), there are 4 access categories (ACs) that directly correspond to the user priority (in order of priority, from lowest to highest):

1. Background (AC_BK)

2. Best Effort (AC_BE)

3. Video (AC_VI)

4. Voice (AC_VO)

There are queues that correspond to each of these access categories. When an MSDU (MAC Service Data Unit) is received from the LLC, it will be placed on the queue that corresponds to its priority. Each queue has an associated set of EDCA parameters whose values determine the probability of transmission. When multiple stations have data to transmit, a station that has EDCA values that correspond to a higher priority will not have to wait as long to transmit as the station sending lower priority data.

The EDCA parameters consist of *Arbitration Inter-Frame Space*, *Transmit Opportunity*, and minimum and maximum values for the *Contention Window*. The meaning of these parameters and their values will not be discussed here, as this is outside the scope of this work.

### *Channel Routing*

In a WAVE system, the EDCA mechanism must be used per channel. For a device that supports alternating channel access, multiple sets of EDCA queues will be needed, one set per channel. The reference architecture shown in Figure 2.3 contains two instances of the 802.11p MAC, one for the control channel and another for a service channel. The 1609.4 MAC extension is responsible for routing data to the appropriate channel, as well as transmitting the data on the correct channel at the correct time. On a WAVE device, there are two types of data frames that can be transmitted:

1. Frames carrying IPv6 datagrams
2. Frames carrying WAVE Short Messages (WSMs)

Data of both types arrives at the WAVE MAC from the LLC in the form of an MSDU. IP data arrives via a MA-UNITDATA request primitive (specified in 802.11), and WSMP data arrives via a MA-UNITDATAX request. The MA-UNITDATAX request is identical to the MA-UNITDATA request, but with the addition of 4 parameters:

1. Channel Identifier
2. Data Rate
3. Transmit Power Level
4. Expiry Time

Once an MA-UNITDATA request or an MA-UNITDATAX request is received, the *Ethertype* field is then used in order to determine the type of the data. For WSMP, the channel, transmit power, and data rate are included in the request. For IP data, these parameters must be retrieved from a transmitter profile, located in 1609.4's Management Information Base (MIB). An access category for the MSDU is assigned by its user priority. After the channel and the access category are determined, MAC protocol information is added to the MSDU, creating a MAC Protocol Data Unit (MPDU). The MPDU is then placed on a queue according to the channel and the access category. Once we reach the channel on which the data will be sent, the MAC waits for a clear channel, then selects an MPDU from the queue that wins internal contention. The MAC then sends a series of primitives to the PHY in order to set the transmit power and data rate, followed by a series of PHY-DATA requests that contain the data to be transmitted. From here, the PHY handles transmission of the data over the physical medium. This process is illustrated in Figure 2.4.

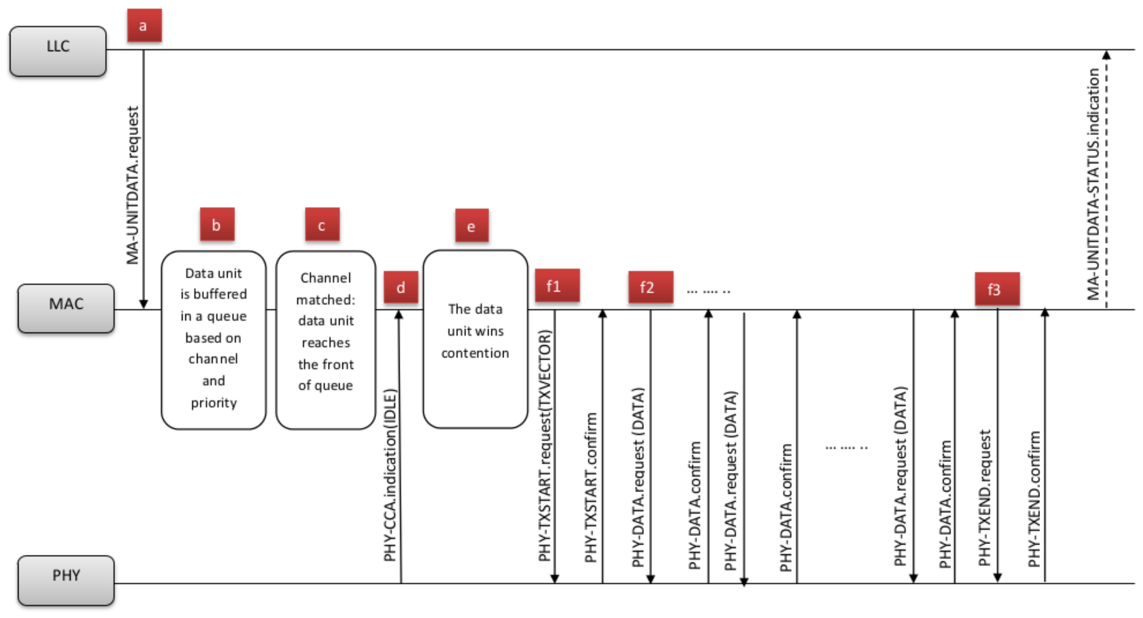


**Figure 2.4**: 1609.4 Transmit Operation [8]

### 2.2.2 Management Services

On the management plane, 1609.4 handles timing synchronization, controlling channel access, handling the transmission and reception of Vendor Specific Action frames, maintaining the Management Information Base, changing the device's MAC address, and access to other 802.11 services on a per-channel basis.

*Timing Synchronization*

If a WAVE device is to provide alternating channel access, then it is mandatory that it is synchronized to UTC time. This can be achieved in multiple ways, through an external time source, such as GPS, or through the use of Timing Advertisements. GPS provides a 1 PPS (Precise Pulse per Second) signal synchronized to UTC time, which can be used to update the device's estimate of UTC time.

For devices that do not have a GPS signal (or other external time source), they are able to synchronize to UTC time using the Timing Advertisement mechanism. A Timing Advertisement user updates its estimate of UTC based on the time sent from a Timing Advertisement provider. The process of transmitting and receiving Timing Advertisements is as follows:

1. 1609.4's MLME Extension receives a request from an upper layer. In this request, the *Channel Identifier*, *Channel Interval*, *Destination MAC Address*, and *Repeat Rate* are included, and *Timing Advertisement Contents* are optionally included.

2. The MLME Extension then generates requests to send Timing Advertisements to the 802.11p MLME. These requests are generated according the specified *Repeat Rate*, and contain the timing information necessary for the receiving device to update its estimate of UTC time.

3. Upon receiving a request from the 1609.4 MLME Extension, the 802.11p MLME transmits a Timing Advertisement frame.

4. The Timing Advertisement frame is received at the lower layers, and the data is passed up to the 1609.4 MLME Extension, which then may generate its UTC estimate using the timing information contained in the frame.

Although it is mandatory that devices are synchronized to UTC time, there still may be some slight timing inaccuracies between devices. In order to account for this, as well as the amount of time it takes the radio to switch channels, there is a 4 ms guard interval at the beginning of each channel interval. This is made up of 2 ms *SyncTolerance* and 2 ms for *MaxChSwitchTime*. The first and last ms of the guard interval are *SyncTolerance*/2. Milliseconds 2 and 3 are *MaxChSwitchTime*. During the *SyncTolerance* period, only receive operation is permitted. During the *MaxChSwitchTime*, no transmit or receive operation is allowed.

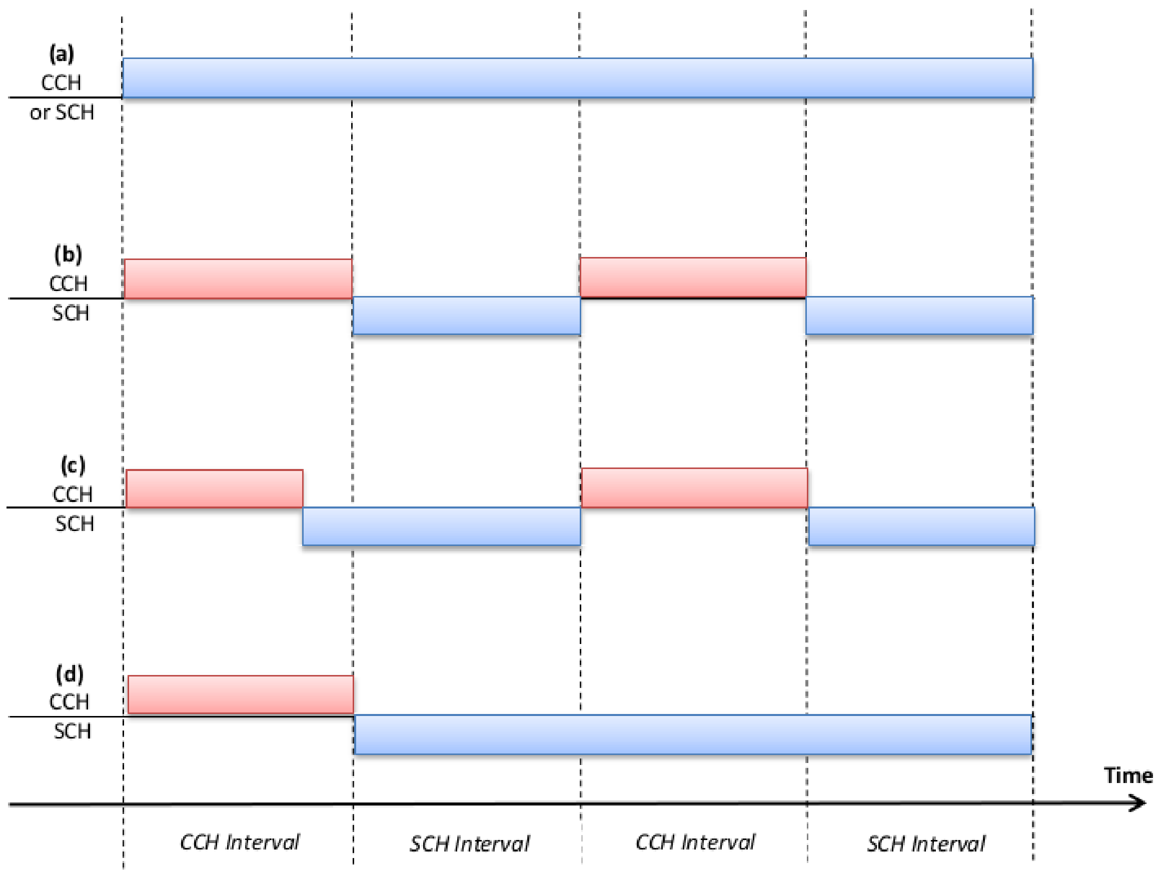


**Figure 2.5**: Channel access modes specified in IEEE 1609.4 [8]

*Channel Access*

In 1609.4, there are two channel types, control channel and service channel. The control channel is used for management and high-priority data. Service channels are used for general application data. There is one control channel available and six service channels. 1609.4 allows 4 channel access modes, shown in Figure 2.5:

(a) Continuous

(b) Alternating

(c) Immediate

(d) Extended

Time is divided into two intervals, the CCH (Control Channel) interval and the SCH (Service Channel) interval. These intervals are each 50 ms long, and together they make up a *Sync Interval*, with the SCH interval following the CCH interval. The beginning of each UTC second starts with a *Sync Interval*, and there must be an integer number of *Sync Intervals* in each UTC second.

The continuous access mode refers to single service channel or control channel, with no channel switching. In the alternating channel access mode, the channel is switched according to the CCH and SCH intervals, e.g., the radio is switched to the control channel during the CCH interval, and switched to a service channel during the SCH interval. In the immediate channel access mode, the service channel is switched to immediately, interrupting the current CCH interval, not waiting for the next SCH interval. The extended access mode allows continuous access to a service channel over a specified amount of time, without switching back to the control channel until the end of the extended access period.

Channel access is initiated upon the receipt of a MLMEX-SCHSTART request from upper layers. This request contains the channel, access mode, a set of EDCA parameters to use for this SCH, and a set of data rates available for use on this channel. Channel access is ended through the use of a MLMEX-SCHEND request.

### *Vendor Specific Action Frames*

1609.4 supports the transmission and reception of Vendor Specific Action frames (VSA frames). These are a type of management frame defined in 802.11, and are used in WAVE to transmit WAVE Service Advertisements (WSAs, defined in 1609.3), as well as other WAVE management data.

Transmission of VSA frames is handled in the same way as Timing Advertisement frames. A request is received from upper layers containing the *Destination Mac Address*, *Repeat Rate*, *Channel Identifier*, *Channel Interval*, as well as fields specific to

VSA frames: *Management ID*, *Organization Identifier*, and *Vendor Specific Content*. The *Management ID* parameter is used to identify the data source for 1609 entities, and the *Organization Identifier* is used for non-1609 entities. Possible values for these fields can be found in the IEEE 1609.0 and IEEE 802.11p standards documents. *Vendor Specifc Content* carries the actual data being sent in the VSA frame.

*Management Information Base*

Maintainence of a Management Information Base (MIB) is mandatory for devices implementing the 1609.4 standard. The MIB contains configuration and status information relating to the 1609.4 entity. The information contained consists of:

1. Capabilities - specifies which 1609.4 features are implemented on this device.
2. Switching - contains lengths of channel and guard intervals
3. Channel Set Table - contains information about the channels that are available for use
4. EDCA CCH Table - contains values of EDCA parameters for the control channel
5. EDCA SCH Table - contains values of EDCA parameters for service channels
6. Transmitter Profile Table - contains values of transmit parameters for IP data on service channels
7. Timing Information - contains information relating to UTC time, whether the device is synchronized, and an estimate of the error

*Readressing*

1609.4 specifies an optional readdressing feature, where the device's MAC address can be changed to a locally administered address in support of pseudonymity. 1609.4's

MLME Extension would receive a request from an upper layer to do this. In the request, a MAC address can be optionally be passed. If there is no MAC address in the request, a random MAC address will be used.

# CHAPTER 3
# IMPLEMENTATION

## 3.1 PLATFORM AND DEVELOPMENT ENVIRONMENT

### 3.1.1 Hardware

The hardware that this implementation runs on is a custom designed Vehicular Multi-technology Communication Device (VMCD) being developed by our partners at tk-LABS. This device contains technologies for both WAVE communications as well as 802.11 WiFi, and cellular data communications. In addition, it includes navigation sensors, an RFID reader, and peripherals such as USB, Ethernet, SD/MMC card, audio I/O and VGA output [25]. At the core of this board is an x86-based processor running embedded Linux.

### 3.1.2 Software

All software on the device runs on top of an embedded Linux operating system. Using Linux on the device has some important advantages [14], the most important of which is the ability to reuse existing software components. With a custom operating system solution, all functionality would have to be built from scratch, requiring a tremendous amount of overhead. With embedded Linux, there are existing drivers for many of the peripherals on the main board, as well as a fully implemented TCP/IP networking stack. This allows us to more efficiently manage our development time and costs, as we will only have to implement new functionality specific to our device.

Linux is also free to use and distribute, as well as being completely open-source. This means that all of Linux's source code is freely available to read and modify as needed. This is another important advantage for the use of Linux over closed-source proprietary operating systems that may cost thousands of dollars to use and distribute.

As part of the Linux operating system, there is an existing implementation of the IEEE 802.11 MAC sublayer, *mac80211*, as well as an open-source driver for our 802.11p chipset, *ath5k*. We took the same approach as [1], and made modifications to these existing kernel modules in order to implement the IEEE 802.11p standard. This work was performed by our partners at tkLABS in addition to the hardware design. tkLABS also implemented features needed in order for our 1609.4 implementation to interface with the *mac80211* and *ath5k* modules, as well as any 1609.4 functionality that was required to be done at the driver level.

### *Development Tools*

A number of open-source and custom tools were used throughout the entire course of the implementation of 1609.4. Since the device has limited resources, development was not done natively; it was done on a development machine, and then the built modules were loaded on to the device for testing. Development was done in a virtual machine (VM) running the x86 version of Ubuntu 12.04 LTS.

The text editor *vim* was used for editing source code files. From [13]:

> Vim is a highly configurable text editor built to enable efficient text editing. It is an improved version of the vi editor distributed with most UNIX systems.

Vim was chosen due to its flexibility and efficiency. It is a terminal-based application (although there is a GUI version available), so that remote files can be edited easily. The development virtual machine was run headless (no GUI) in order to conserve resources on the host machine, so all development was done over ssh (Secure Shell protocol) using the *open-ssh* program running in a terminal. Vim was configured to provide syntax highlighting, show line numbers and plugins were used to provide an in-window file system explorer, auto-complete, and *jump to function definition* functionality. Vim's split window functionality was also used extensively in order to edit multiple files simultaneously.

Proprietary utilities were used for building the kernel and modules and for version control of source files. These utilities were provided by tkLABS. In order to load modules for testing throughout development, *nfsroot* was used. NFS stands for Network File System, and allows for a machine to mount a remote file system and access files as if they are present locally [15]. In our development environment, the development VM is the NFS server, providing a local directory that will be used by the client, our VMCD device. In this setup, the client's root file system is located on the development machine. Once the modules are built on the development VM, the .ko files can be copied to the NFS directory and they will be seen on the device. The reason for this type of setup is so we will not have to remove the device's flash card and reboot every time we want to load a new version of a module.

## 3.2 APPROACH

The 1609.4 functionality was implemented as a kernel module, rather than in user space, as it is fairly low in the protocol stack (close to the PHY layer) and because of its interaction with the existing *mac80211* and *ath5k* kernel modules. The implementation for the 1609.4 module is located in *external_modules/net/1609.4/*, with a header file located in *external_modules/include/net/* that declares publicly accessible functions, constants and data types.

The primitives specified in IEEE 1609.4 were grouped together according to functionality, and entities were defined to implement them. These primitives are as follows:

1. MLMEX-TA

2. MLMEX-TAEND

3. MLMEX-VSA
4. MLMEX-VSAEND
5. MLMEX-SCHSTART
6. MLMEX-SCHEND
7. MLMEX-REGISTERTXPROFILE
8. MLMEX-DELETETXPROFILE
9. MLMEX-CANCELTX
10. MLMEX-GETUTCTIME
11. MLMEX-SETUTCTIME
12. MLMEX-AddressChange
13. MLMEX-SendPrimitive
14. MA-UNITDATAX

Entities defined to implement these primitives are:

**Timing Advertisement Manager** This implements the primitives MLMEX-TA and MLMEX-TAEND, which start and stop the transmission of Timing Advertisements.

**VSA Manager** This implements the MLMEX-VSA and MLMEX-VSAEND primitives, which start and stop the transmission of Vendor Specifc Action frames. Not implemented as part of this work.

**Channel Scheduler** This implements the MLMEX-SCHSTART and MLMEX-SCHEND primitives, which start and stop providing service channel access.

**MIB** This maintains the Management Information Base, and implements the following primitives:

- MLMEX-REGISTERTXPROFILE
- MLMEX-DELETETXPROFILE
- MLMEX-GETUTCTIME
- MLMEX-SETUTCTIME

These primitives are responsible for registering and deleting transmitter profiles, and accessing/modifying the UTC time estimate.

There were no separate entities defined to implement the MLMEX-CANCELTX, MLMEX-AddressChange, MLMEX-SendPrimitive, and MA-UNITDATAX primitives. MLMEX-CANCELTX cancels all transmissions on a particular queue, given a channel number and access category. Since this only requires a single function call from 1609.4, it is implemented in the top-level entity. The MLMEX-AddressChange primitive is used to change the MAC address of the device, for pseudonymity. This optional functionality is available from userspace in Linux, but not from kernelspace. This was not implemented as part of this work. MLMEX-SendPrimitive allows services from 802.11 to be called from layers above 1609.4. This primitive is also optional, and is not implemented as part of this work. Access to any 802.11 services will be added as needed. The MA-UNITDATAX primitive is used to pass WSMP data from the LLC to the MAC. This is identical to the MA-UNITDATA primitive used for passing IP data, but with the addition of 4 transmission parameters. Because of this, a new entity was not defined; the existing MA-UNITDATA functionality was modified to handle both IP and WSMP data.

The overall organization of these entities uses the object-oriented design elements of abstraction and encapsulation. Each one of these entities is equivalent to a class, consisting of a private set of data that can only be accessed or modified through the use of public functions. This keeps implementation details hidden from users of these entities. The 1609.4 top-level entity encapsulates all of these entities, hiding their existence from code that uses 1609.4. Figure 3.1 shows a simplified class diagram of the implementation.

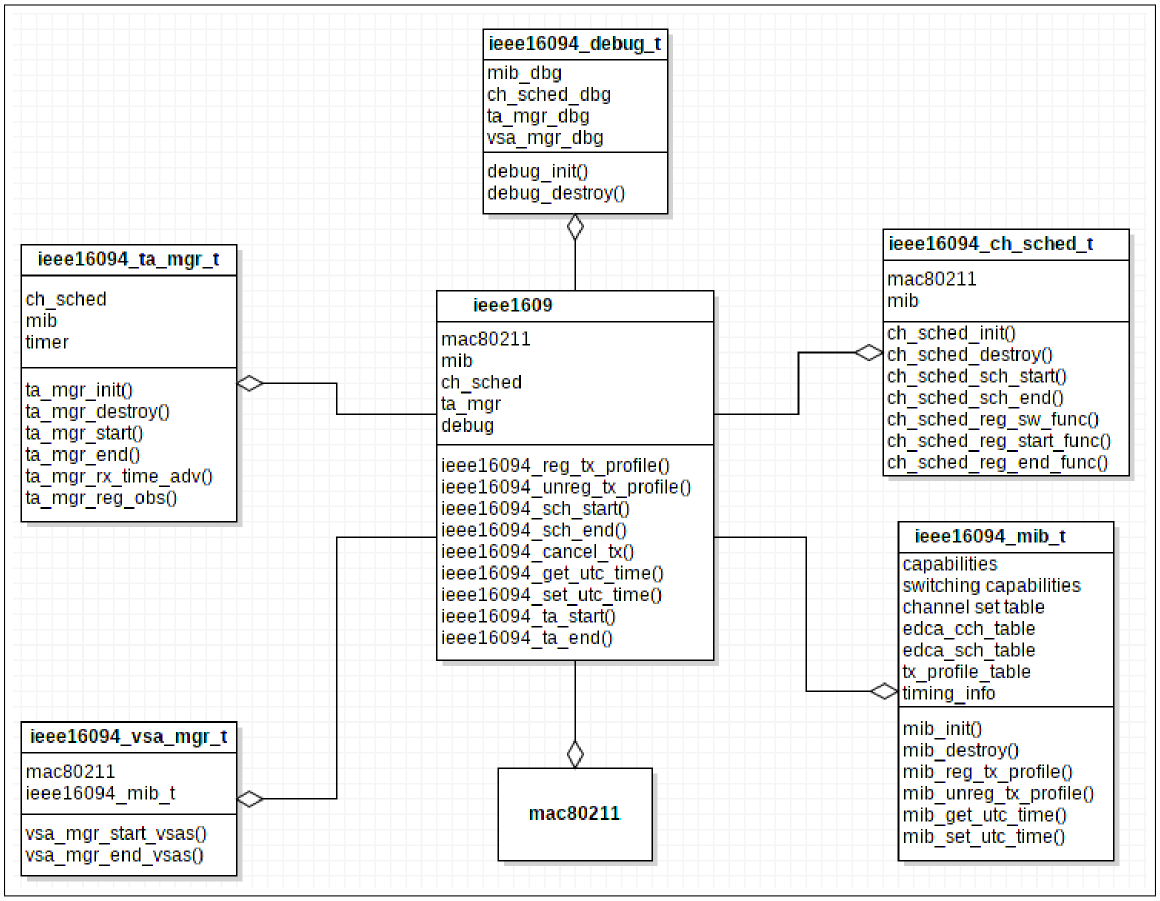


**Figure 3.1**: Class diagram showing the structure of 1609.4 entities

In order to acheive the data hiding and abstraction associated with object-oriented languages, such as C++ or Java, the concept of opaque pointers was used. Structs representing the entities, as well as functions used to manipulate them, are defined in .c files, while the public functions are declared in the corresponding .h files. Also declared in these .h files are the struct types, with no public definition. This allows a pointer to this struct type to be declared and passed to functions, without providing access to its internal data. The majority of these entities are used by the top-level

1609.4 entity in the following way:

1. A pointer to the struct type representing the entity is declared.
2. The pointer is initialized and allocated by calling the entity's *init* function.
3. The entity's public functions are called to access or modify data belonging to the entity. These functions require that an allocated pointer to the struct type is passed as an argument.
4. When the entity will no longer be used, it is de-allocated by calling its *destroy* function.

The top-level entity maintains pointers to each of these struct types in order to pass to functions as needed. The top-level entity is represented by the struct *ieee1609*, which is declared and defined in *external_modules/net/1609.4/ieee16094.h*. The fields of this struct are visible to other files within the 1609.4 implementation, but not to other modules. A pointer to this struct is declared globally in order to access it from the module initialization and exit functions. These are functions to be called at the time of loading and unloading the module [16]. Neither of these functions accept parameters, so any data that needs to be accessed here must be globally accessible.

The top-level 1609.4 entity is initialized when the kernel module is loaded. This initialization function calls all of the initialization functions for each of the contained entities. After the module is loaded, the management functions can be called by other modules, most likely from 1609.3's WME. These functions, in turn, call functions from each of the entities that actually implement the primitives. This is to provide a consistent interface to other modules. All of these functions correspond to the management primitives specified in the standard. For the data plane primitive, MA-UNITDATAX, there is no public function in 1609.4 that corresponds to this, instead it is called through the *mac80211* module, from the LLC.

## 3.3 DATA PLANE

The Data Plane of 1609.4 consists of the *Channel Coordination*, *Channel Routing*, and *User Priority* services. Together, these services describe the transmit and receive operation for data transmission. The only two data plane primitives exchanged with an upper layer are the MA-UNITDATA and MA-UNITDATAX primitives. These are used to pass IPv6 data or WSMP data from the LLC to the MAC. Since IPv6 already exists in the Linux kernel, the MA-UNITDATA primitive did not need to be implemented. However, the MA-UNITDATAX primitive is identical to the MA-UNITDATA primitve, but with the addition of the transmit parameters. In order to implement the MA-UNITDATAX request, the decision was made to modify the implementation of the MA-UNITDATA request in order to be able to handle both IP and WSMP data. In order to effectively describe this implementation, some background information on Linux networking will first be provided.

### 3.3.1 Linux Networking Background

In the Linux kernel, a data packet to be transmitted is stored in an instance of the *struct sk_buff*, or socket buffer [17]. All networks layers use this struct in order to store headers and data to be transmitted. As an *sk_buff* to be transmitted is passed down from one layer to the next, the new layer adds space to the beginning of the buffer and places its header.

*sk_buff* contains a number of fields, relating to its layout and structure, as well as many feature-specific fields. In-depth descriptions of every field are beyond the scope of this work. However, an important field to mention is the *cb* field, or control buffer. This is a fixed-size field of 40 bytes that is available for private use by each layer. The control buffer is used by the *mac80211* module to pass transmission information to the driver.

Another important struct is *net_device*. This represents a network device (such as a Network Interface Card (NIC)), and contains related configuration information. Information about the interface type is contained here, and can be used, for example, to differentiate between a WAVE device and a WiFi device.

*Receiving Data From The LLC Sublayer*

The Logical Link Control (LLC) sublayer implementation is located in the directory */net/llc/*. The function *llc_build_and_send_ui_pkt()* in *llc_output.c* calls a sequence of functions that eventually reach the *mac80211* module, carrying an *sk_buff* to be transmitted. This sequence of function calls starts with *dev_queue_xmit()*, which eventually (indirectly) calls a function pointer, *ndo_start_xmit()*. This function pointer is part of a set of function pointers associated with a net device, *net_dev_ops*. The pointers here are initialized by a driver or lower level module. In this case, *ndo_start_xmit()* points to a function in the *mac80211* module. This function, *ieee80211_subif_start_xmit()*, implements the MA-UNITDATA request. The parameters passed include an *sk_buff* and a *net_device*. All of the parameters associated with the MA-UNITDATA request are located within one of these two structs.

*ieee80211_tx_info struct*

As mentioned previously, the *sk_buff* struct contains a 40-byte control buffer field, *cb* that can be used for private data among layers. The *mac80211* module uses this field for passing transmit information to the driver [18]. When an *sk_buff* is received from the LLC through the call *ieee80211_subif_start_xmit()*, a pointer to a *ieee80211_tx_info* struct is declared, and initialized to point to the control buffer. The control buffer's original contents are erased, and the control buffer is filled with the contents of the *ieee80211_tx_info* struct. When the *sk_buff* reaches the driver, all of the information added to the control buffer can be accessed by casting the *cb* field to the *ieee80211_tx_info* type.

### 3.3.2 Implementation of MA-UNITDATAX

As previously mentioned, the MA-UNITDATA request is already implemented in the *ieee80211_subif_start_xmit()* function. This function was modified so that it is able to handle the MA-UNITDATAX request as well. Since the modified *mac80211* module falls under the 802.11p implementation done by tkLABS, the implementation of the MA-UNITDATAX primitive includes some assistance from tkLABS in addition to our code.

As part of the 802.11p implementation, an interface type was added for use with WAVE devices: *NL80211_IFTYPE_WAVE*. If this is the interface type associated with the *net_device*, then code specific to WAVE is executed, including a call to 1609.4 functionality needed for the MA-UNITDATAX request primitive. In order to call 1609.4 functionality from *mac80211*, the *ieee80211_1609dot4_ops* struct was created, containing pointers that would be called by the *mac80211* module when 1609.4 functionality would be needed. The fields of this struct are as follows:

1. **int (*tx_prepare)(struct sk_buff *skb);**
2. **int (*rx_time_advertisement)(struct ieee80211_mgmt *mgmt);**
3. **int (*rx_vendor_specific_action)(struct ieee80211_mgmt *mgmt);**
4. **int (*get_tx_rate)(struct sk_buff *skb);**

In the 1609.4 module's *main.c*, an instance of this struct is defined, and the pointers are initialized to the appropriate 1609.4 functions. These functions are then registered with the *mac80211* module by calling *ieee80211_register_1609dot4()*, passing the address of the struct. In implementing the MA-UNITDATAX request, the *tx_prepare()* function pointer is called from *ieee80211_subif_start_xmit()* if the interface type indicates WAVE.

Also modified was the *mac80211* module's *ieee80211_tx_info* struct. A new struct, *ieee16094_params*, was defined to store the four additional transmit parameters used in 1609.4. An instance of this struct was added to *ieee80211_tx_info* and is used to pass these parameters to the driver. The fields of *ieee16094_params* are as follows:

1. **u8 channel_index;**
2. **u8 data_rate;**
3. **u8 tx_pwr;**
4. **u64 expiry_time;**

The function pointer *tx_prepare()* points to the function *ieee1609_tx_prepare()*, which is defined in *main.c*. This function is responsible for populating the fields of the *ieee16094_params* struct for data being transmitted. The general functionality is as follows, and is shown in Figure 3.2.

1. Check the value of the *Ethertype* header.
   (a) If *Ethertype* indicates WSMP, parse WSMP header to obtain values of transmit parameters.
   (b) If *Ethertype* indicates IP, retrieve transmit parameter values from MIB.
2. Validate parameters and copy values to the *ieee16094_params* struct located in the control buffer.

The remainder of 1609.4's transmit operation takes place at the driver level, and is implemented by tkLABS in the *ath5k* driver. The struct *ieee16094_params* is used by the driver in order to place the *sk_buff* in a queue corresponding to its channel and priority, and set the transmit power and data rate prior to transmission. The EDCA/user priority mechanism already exists as part of 802.11, though a set

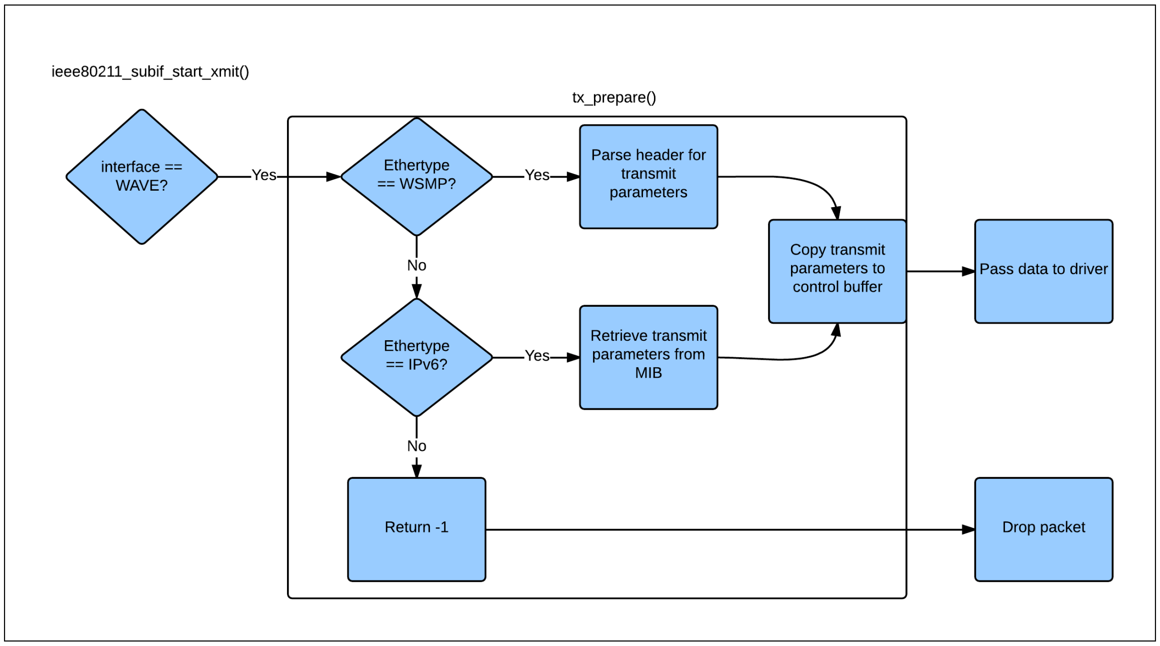

ieee80211_subif_start_xmit()
tx_prepare()
interface == WAVE?
Yes
Ethertype == WSMP?
Yes
Parse header for transmit parameters
No
Copy transmit parameters to control buffer
Pass data to driver
Ethertype == IPv6?
Yes
Retrieve transmit parameters from MIB
No
Return -1
Drop packet

**Figure 3.2**: MA-UNITDATAX Implementation

of software queues per channel was implemented in the *ath5k* driver. While the 1609.4 module selects the current channel, and associates a channel number with an *sk_buff*, the *ath5k* driver places the *sk_buff* in the appropriate queue, and also handles transmission of the *sk_buff* from the queue that corresponds to the current channel.

### 3.3.3 Reception of Data Frames

In 1609.4, the reception of both WSMP frames and IPv6 frames is handled by the MA-UNITDATA indication primitive. This is already implemented by the Linux kernel, and needed no modification. The upper layers, when passed data received from the MAC, can distinguish between the two types of data by using the *Ethertype* header field.

## 3.4 MANAGEMENT PLANE

### 3.4.1 Channel Access and Switching

In this implementation of IEEE 1609.4, the channel access mode defaults to continuous access on the control channel. The channel access mode is changed upon receipt of the MLMEX-SCHSTART request or MLMEX-SCHEND request primitives. In order to manage channel access, a *Channel Scheduler* entity was implemented. This is implemented in the file *channel_scheduler.c*. The *Channel Scheduler* supports all four channel access modes: continuous, alternating, immediate, and extended. In addition to these four access modes, we also support alternating access between the control channel and multiple service channels. This section describes the need for multiple concurrent service channel access, as well as the implementation of the *Channel Scheduler*, including the structure and algorithm used.

#### *Multiple Concurrent Service Channel Access*

Although there are six available service channels, all of the examples shown in the standard show alternating access between one service channel and the control channel. The standard does not specify how to handle the possibility of a WAVE device concurrently participating in multiple services. One of the challenges here is hardware limitations. Our 802.11p wireless chipset supports 10 hardware queues. According to the standard, the EDCA mechanism must be used per-channel, which requires a queue for each access category. This means that in order to support alternating access with all six service channels and the control channel, we would require four queues for each of the seven channels, for a total of 28 queues. Since we are limited to 10 hardware queues, this would require the implementation of software queues. Prior to implementing these, we discussed a possible real-world scenario that would necessitate the use of multiple concurrent service channel access.

In WAVE, service channels can be designated for specific purposes. For example, channels 172 and 184 are designated for public safety applications. From IEEE 1609.0:

> Channels 172 and 184 are designated for public safety applications involving safety of life and property. Specifically, channel 172 is for "vehicle-to-vehicle safety communications for accident avoidance and mitigation, and safety of life and property applications;" Channel 184 is for "high-power, longer distance communications to be used for public safety applications involving safety of life and property, including road intersection collision mitigation."

Other service channels may be designated to serve specific purposes, for example, weather, traffic, or infotainment. An example real world scenario that illustrates the need for multiple concurrent SCH access is as follows:

1. A single-radio device needs to continuously broadcast updated traffic information on SCH1, as well as safety information on SCH2, and real-time weather updates on SCH3.

2. We may continuously receive small amounts of data from the application layer, that can all be transmitted in one SCH interval.

3. When this data is passed down to the 1609.4 layer, it will need to be queued until this particular SCH interval occurs, when it can be transmitted.

For this scenario, with a device operating on three SCHs (and the CCH) concurrently, a set of four queues (one for each access category) would be needed for each channel, resulting in a total of 16 queues. For a similar scenario with all six service channels, plus the control channel, we would need 28 queues. Since this queueing takes place at the lower MAC and PHY layers, the implementation was handled by tkLABS.

*Public Functionality*

The *Channel Scheduler* is represented by the struct *ieee16094_ch_sched_t*, which is defined in the file *channel_scheduler.c*. There is a public declaration of this type in *channel_scheduler.h*, allowing users of the *Channel Scheduler* to reference it via an opaque pointer. The reasoning for this is to hide implementation details from calling code, and to ensure that fields used internally are not able to be modified directly. Users can only access and modify the *ieee16094_ch_sched_t* through publicly available functions declared in *channel_scheduler.h*. These include functions to create and destroy a *Channel Scheduler*, functions that correspond to the channel access primitives, functions to register observers (explained in this section), accessor functions for certain fields, and channel-related utility functions.

Only the functions important to the functionality of the *Channel Scheduler* will be described here, descriptions of the accessors and utility functions will be left out. The most important functions are as follows:

1. **struct ieee16094_ch_sched_t *ch_sched_init(struct ieee16094_mib_t *mib);**

2. **void ch_sched_destroy(struct ieee16094_ch_sched_t *ch_sched);**

3. **int ch_sched_sch_start(struct ieee16094_ch_sched_t *ch_sched, u8 ch_num, const u8 *rates, u8 num_rates, const struct edca_set_t *edca_params, bool immediate_access, u8 extended_access);**

4. **int ch_sched_sch_end(struct ieee16094_ch_sched_t *ch_sched, u8 ch_num);**

The *ch_sched_init()* function allocates memory for the *Channel Scheduler*, initializes all of its fields, and returns a pointer to a *ieee16094_ch_sched_t* representing the new *Channel Scheduler*. This is called when the 1609.4 module is loaded. The resulting pointer must be passed to any other *Channel Scheduler* functions to be called. The *ch_sched_destroy()* function frees all memory allocated for the *Channel Scheduler*, and is called when the 1609.4 module is unloaded. The functions *ch_sched_sch_start()* and *ch_sched_sch_end()* correspond to the MLMEX-SCHSTART request and MLMEX-SCHEND request primitives, respectively.

Since the *Channel Scheduler* depends on the MIB, a pointer to a previously initialized *ieee16094_mib_t* is passed to the initialization function. The *ch_sched_sch_start()* takes the parameters specified for the MLMEX-SCHSTART request primitive. These are as follows:

1. **ch_num** - The service channel to provide access to
2. **rates** - A set of data rates that can be used on the specified channel
3. **num_rates** - The number of data rates in **rates**
4. **edca_params** - A set of EDCA parameters to be used for communication on the specified channel
5. **immediate_access** - If this is true, service channel access should be provided immediately rather than waiting until the next SCH interval
6. **extended_access** - If this is greater than zero, service channel access will be provided during both the SCH interval and the CCH interval, for the specified number of CCH intervals

In addition to these, there is a set of functions used in order to notify users of the *Channel Scheduler* that channel access has started, stopped, or that the channel has been switched. These functions are used to register a function to be called by the *Channel Scheduler* in response to the previously mentioned events:

1. **int ch_sched_reg_sw_func(struct ieee16094_ch_sched_t *ch_sched, void (*func)(bool, enum access_mode_t, enum wave_ch_num_t, struct timeval *));**

2. **int ch_sched_reg_start_func(struct ieee16094_ch_sched_t *ch_sched, void (*func)(bool, u8, enum wave_ch_num_t, struct timeval *));**

3. **int ch_sched_reg_end_func(struct ieee16094_ch_sched_t *ch_sched, void (*func)(struct timeval *));**

The use of these functions somewhat models the *observer* object-oriented design pattern. The observer pattern allows objects referred to as *observers* to react to events of another object, the *subject*, without requiring the subject to be dependent on the observers [19]. Our implementation, although written in C, a procedural language, is organized in an object-oriented manner. The *Channel Scheduler* acts as the subject, and observers are the *Timing Advertisement Manager* and the *Channel Scheduler Debug* entities.

The struct *ieee16094_ch_sched_t* maintains 3 arrays of function pointers which are called when the channel is switched, channel access is started, or channel access is stopped:

1. **void (*notify_ch_sw[MAX_OBSERVERS])(bool, enum access_mode_t, enum wave_ch_num_t, struct timeval *);**

2. **void (*notify_sch_end[MAX_OBSERVERS])(bool, u8, enum wave_ch_num_t, struct timeval *);**

3. **void (*notify_sch_end[MAX_OBSERVERS])(struct timeval *);**

An observer uses the previously mentioned registration functions in order to assign one of its functions to one of these function pointers. By using this approach rather than calling an observer's function directly, the *Channel Scheduler* will not need to be modified due to changes in one or more of its observers.

### *Channel Switching*

In 1609.4, channel switching is done at strict 50 ms intervals, with time being split up into 50 ms Control Channel (CCH) intervals and 50 ms Service Channel (SCH) intervals. The beginning of every UTC second starts with a CCH interval, followed by an SCH interval 50 ms later, and this repeats until the start of the next UTC second, which begins with another CCH interval. Due to the extreme importance of the accuracy for this 50 ms channel switching, tkLABS implemented a 50 ms timer for us that compensates for drift, as they are domain experts in this area. This timer is implemented in the file *tick.c*, using the Linux kernel's high resolution timer APIs. High resolution timers will be discussed more in depth in the Timing Advertisements section.

The code in *tick.c* was modified in order to add a callback to other entities when the 50 ms timer expires, as well as a function that can be used to assign a function to this new function pointer. This allows the *Channel Scheduler* to receive notifications on channel interval boundaries, without requiring the code in *tick.c* to be aware of any of the *Channel Scheduler*'s code. This works in the same way as the observer pattern described above.

The *Channel Scheduler* controls channel access and switches channels based on the current channel access mode. When it receives the callback from *tick.c*, the channel can be switched if appropriate, given the access mode and channel interval.

### *Algorithm*

When the *Channel Scheduler* is initialized, it registers a function, *handle_ch_interval()*, that will be called by the timer from *tick.c*. In addition to this, the fields of the *ieee16094_ch_sched_t* are initialized, and the control channel is switched to, as the default access mode is continuous CCH.

When a MLMEX-SCHSTART request is received from upper layers, an access mode is assigned based on the values of the *immediate_access* and *extended_access* parameters . The possible values and corresponding access modes are as follows:

1. *immediate_access* = false, *extended_access* = 0

   access_mode = CH_ACCESS_ALTERNATING

2. *immediate_access* = false, $1 \leq$ *extended_access* $< 255$

   access_mode = CH_ACCESS_EXTENDED

3. *immediate_access* = false, *extended_access* = 255

   access_mode = CH_ACCESS_CONTINUOUS

4. *immediate_access* = true, *extended_access* = 0

   access_mode = CH_ACCESS_IMMEDIATE

5. *immediate_access* = true, $1 \leq$ *extended_access* $< 255$

   access_mode = CH_ACCESS_EXTENDED

6. *immediate_access* = true, *extended_access* = 255

   access_mode = CH_ACCESS_CONTINUOUS

Although this is not explicity mentioned in the standard, it is important to note that, since *immediate_access* and *extended_access* are separate parameters, it is possible to have both immediate channel access and extended channel access, or both immediate channel access and continuous channel access. However, additional access modes were not defined for these cases, as they are unnecessary for handling these, but there was an additional access mode defined for the case when we have multiple concurrent service channel access, *CH_ACCESS_MULT_SCH*.

The *Channel Scheduler* also contains an array of active channels that correspond to the service channels for which access is being provided. If there is no access being provided to any service channels, then this contains the control channel. Figure 3.3 shows the general functionality, while the specific algorithms for each access mode are described in the remainder of this section.

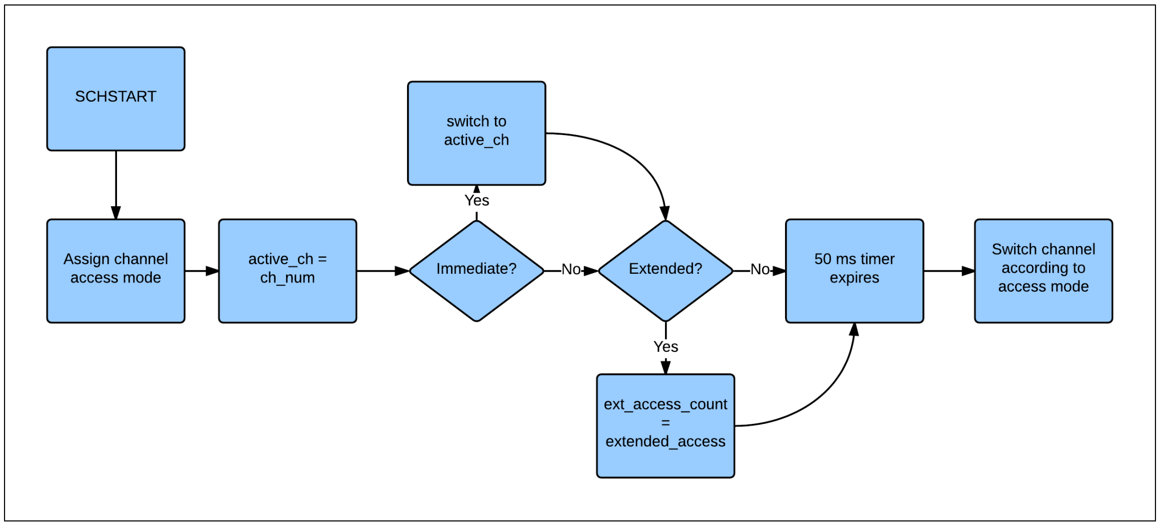

SCHSTART
Assign channel access mode
active_ch = ch_num
Immediate?
Yes
switch to active_ch
No
Extended?
No
Yes
ext_access_count = extended_access
50 ms timer expires
Switch channel according to access mode

**Figure 3.3**: Channel Scheduler Implementation

For continuous channel access, the algorithm is as follows:

1. An MLMEX-SCHSTART request is received with *extended_access* = 255.

2. The *Channel Scheduler*'s access mode is assigned the value of CH_ACCESS_CONTINUOUS, and its active channel is assigned the value of the channel number received in the request.

3. If *immediate_access* is true, the channel is switched to the active channel. If not, we wait for the next SCH interval.

4. We get a callback notifying us that we have changed channel intervals. This

callback provides a boolean value, *cch*, that lets us know if we have entered a CCH interval or an SCH interval.

5. If we have entered an SCH interval, switch to the active channel, else wait for next callback and repeat previous step.

6. Stay on this service channel until the receipt of the MLMEX-SCHEND request primitive.

For alternating channel access (with a single service channel), the algorithm is as follows:

1. An MLMEX-SCHSTART request is received with $immediate\_access = 0$, and $extended\_access = 0$.

2. The access mode is assigned to CH_ACCESS_ALTERNATING, and the active channel is assigned to the channel number received in the request.

3. On all following channel intervals:

    (a) If the interval is a CCH interval, switch to the control channel.

    (b) If the interval is an SCH interval, switch to the active service channel.

For immediate channel access, the algorithm is as follows:

1. An MLMEX-SCHSTART request is received with $immediate\_access = 1$, and $extended\_access = 0$

2. The channel access mode is assigned the value of CH_ACCESS_IMMEDIATE, and the active channel is assigned the value of the channel number received in the request.

3. The channel is switched immediately to the active channel, regardless of whether we are currently in a CCH or an SCH interval.

4. On the next channel interval, the channel access mode is changed to CH_ACCESS_ALTERNATING, and the channel is switched as described in the algorithm for alternating channel access.

5. For all subsequent channel intervals, the algorithm for alternating channel access is followed.

For extended channel access, the algorithm is as follows:

1. An MLMEX-SCHSTART request is received with $\mathit{immediate\_access} = 0$ and $1 \leq \mathit{extended\_access} < 255$

2. The access mode is assigned the value CH_ACCESS_EXTENDED, and the active channel is assigned the value of the channel number received in the request. The *ieee16094_ch_sched_t*'s field *extended_access_count* is assigned the value of *extended_access* from the request.

3. At the next SCH interval, switch to active channel.

4. For subsequent CCH channel intervals, decrement *extended_access_count*.

5. When *extended_access_count* reaches 1, the channel access mode is chaned to CH_ACCESS_ALTERNATING

6. For all subsequent channel intervals, the algorithm for alternating channel access is followed.

In addition to these, we support multiple concurrent service channel access, implemented in a round-robin fashion. A separate access mode was defined for this, since it is a special case. Multiple concurrent SCH access is initiated by the receipt of an MLMEX-SCHSTART request for a second service channel, while we are currently in the alternating channel access mode. The algorithm for initiating multiple concurrent SCH access is as follows:

1. Starting with alternating access for a single service channel and the control channel, an MLMEX-SCHSTART request is received containing the channel number for a second service channel. *immediate_access* = 0 and *extended_access* = 0.

2. The channel access mode is changed from CH_ACCESS_ALTERNATING to CH_ACCESS_MULT_SCH, and the channel number in the request is added to the array of active channels.

3. On all following channel intervals:

   (a) If the interval is a CCH interval, switch to the control channel.

   (b) If the interval is an SCH interval, increment the active channel index, and switch to the new active channel. If active channel index is at the last channel in the array, reset to 0 instead of incrementing.

4. New MLMEX-SCHSTART requests will add the new service channel to the array of active channels.

5. When an MLMEX-SCHEND request is received, the channel specified in the request is removed from the array of active channels, and once there is only a single active service channel, the channel access mode will be changed to CH_ACCESS_ALTERNATING.

### 3.4.2 Timing Advertisements

Timing Advertisements are started and stopped using the MLMEX-TA and MLMEX-TAEND primitives. These primitives are implemented in the file *time_adv_mgr.c*. This section will describe how a single TA frame is sent, how a TA frame is received, how the implementation is structured, the APIs relied on, and the algorithm used.

*Sending a Timing Advertisement Frame*

According to the IEEE 1609.4 standard, the 1609.4 MLMEX receives a MLMEX-TA.request from upper layers to start transmission of TA frames. Upon receiving this request, the 1609.4 MLMEX begins to generate and send MLME-TIMING_ADVERTISEMENT request primitives to the 802.11p MLME. According to the standard, the *Time Advertisement* information element is passed as one of the parameters included in this request.

The *Time Advertisement* information element contains the following fields:

1. *Timestamp* - The value of the transmitting device's Timing Synchronization Function (TSF) timer (defined in 802.11)
2. *Timing Capabilities* - The source of the device's UTC time
3. *Time Value* - This is the offset to the *Timestamp* to calculate UTC time
4. *Time Error* - The standard deviation of error of the device's UTC estimate

As mentioned in [12], there is some time that passes between when a TA frame is generated at the 1609.4 MLMEX and when it is transmitted at the lower MAC. Because of this, the authors of [12] populate the *Timestamp* field at the lower MAC instead, where it is closer to being transmitted. Our implementation follows this approach, as we want the time being sent over the air to be as accurate as possible.

At the 802.11p level, we have a function *ieee80211_send_timing_advertisement()* that takes 4 parameters:

1. **struct ieee80211_vif *vif** - A pointer to the virtual 802.11 interface, this is needed for most *mac80211* functions
2. **enum ieee80211_timing_capabilities timing_capabilities** - This is the *Timing Capabilities* field of the *Time Advertisement* information element described above

3. **u8 *dest_mac** - The MAC address of the device that this TA frame is being sent to

4. **u8 *time_error** - This is the *Time Error* field of the *Time Advertisement* information element described above

The *Timestamp* and *Time Value* fields of the *Time Advertisement* information element are handled by the *mac80211* module. The other parameters are set as follows: *timing_capabilities* is set to the value *WLAN_TIMING_TIMESTAMP_OFFSET_UTC* (equal to a value of 2) if we have GPS time available, or to the value *WLAN_TIMING_NO_STD_EXTERNAL_SOURCE* (equal to a value of 0) if we do not have GPS time available. *dest_mac* is set to the value received in the request. The *time_error* parameter is set to a value of all 1's to indicate that it is unknown.

Although management frames are typically queued in the same manner as data frames (with the exception of always being placed in the highest priority queue), we have taken a different approach with our implementation. This approach will be described in detail in the upcoming sections. When a single TA frame is to be sent, it bypasses the queue and is sent immediately due to the sensitivity of timing and synchronization related data. If the frame waits in a queue, then the data contained in the frame will be inaccurate. Figure 3.4 shows transmission of TAs where the timing information is calculated by 1609.4 and the TAs are queued prior to transmission. Figure 3.5 shows the algorithm used in this implementation.

### *Reception of Timing Advertisement Frames*

According to the standard, upon the receipt of a Timing Advertisement frame (if supported), the 1609.4 MLMEX must generate an MLMEX-TA.indication primitive containing the information received in the TA frame. The standard also states that information received in the TA frame *may* be used to generate an estimate of UTC

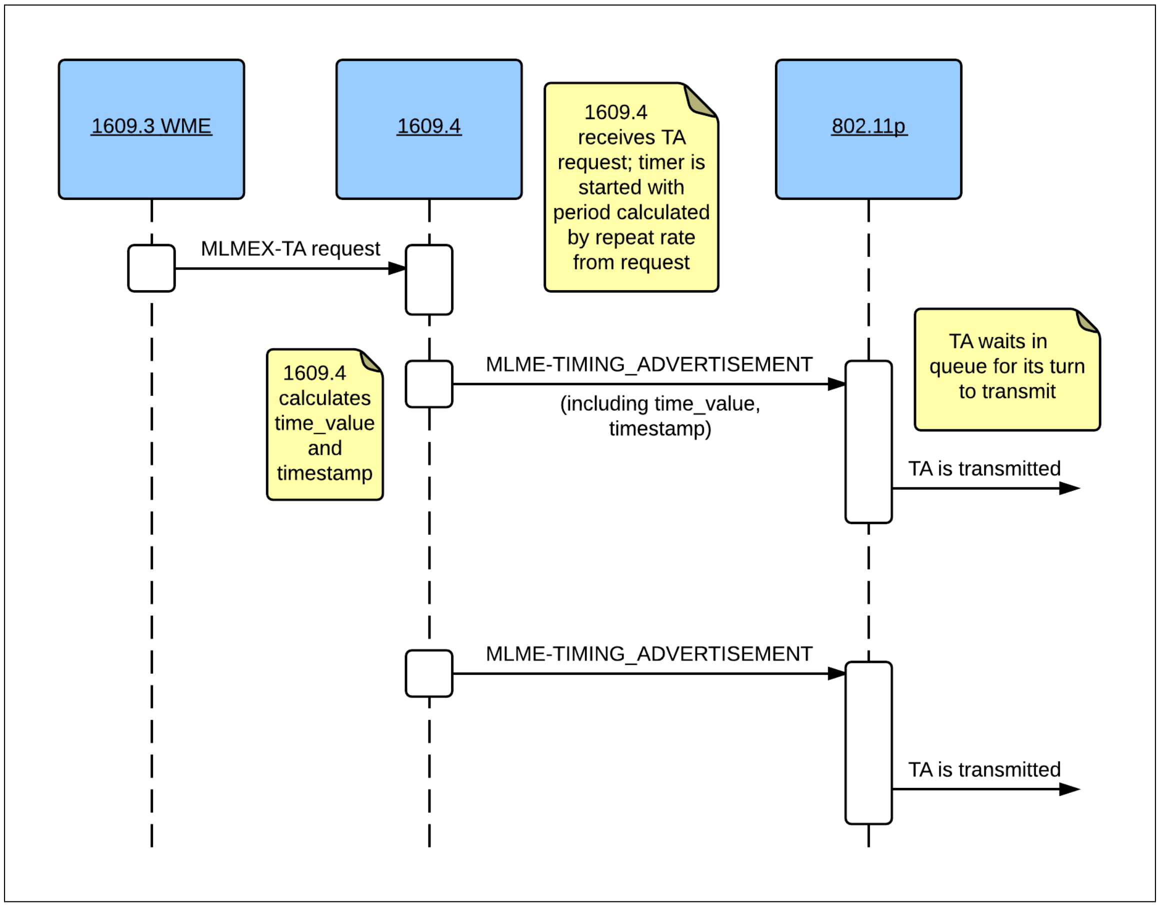

1609.3 WME
1609.4
1609.4 receives TA request; timer is started with period calculated by repeat rate from request
802.11p
MLMEX-TA request
TA waits in queue for its turn to transmit
1609.4 calculates time_value and timestamp
MLME-TIMING_ADVERTISEMENT
(including time_value, timestamp)
TA is transmitted
MLME-TIMING_ADVERTISEMENT
TA is transmitted

**Figure 3.4**: Transmission of TAs where the timestamp and time value are calculated by 1609.4, and the TAs are queued

time. It is not completely clear if we should set the system time in 1609.4's MLMEX, or if the data should just be passed up to 1609.3's WAVE Management Entity to process. Since we do not have a complete implementation of IEEE 1609.3, the decision was made to set the system time in the 1609.4 module. This is done using the function *mib_set_utc_time()*.

*Algorithm for Transmission of Timing Advertisements*

Timing Advertisements are sent according to a repeat rate that is provided in the MLMEX-TA request. This repeat rate indicates the number of TA frames to be transmitted per 5 seconds. However, the standard does not indicate how this should be handled with alternating channel access. The request also contains the channel

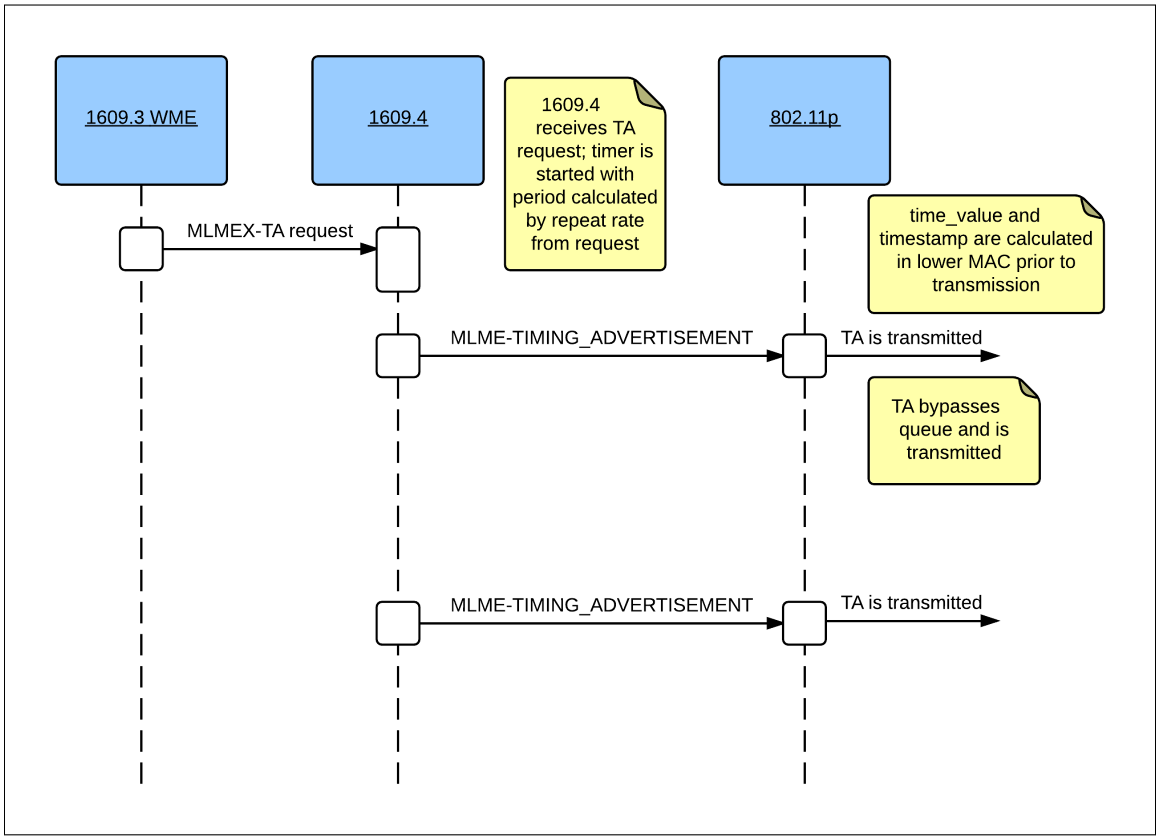

1609.3 WME
1609.4
1609.4 receives TA request; timer is started with period calculated by repeat rate from request
802.11p
time_value and timestamp are calculated in lower MAC prior to transmission
MLMEX-TA request
MLME-TIMING_ADVERTISEMENT
TA is transmitted
TA bypasses queue and is transmitted
MLME-TIMING_ADVERTISEMENT
TA is transmitted

**Figure 3.5**: Transmission of TAs as done in this implementation

that the frames should be sent on, as well as the channel interval during which to send them.

For continuous channel access, the algorithm is simple:

1. An MLMEX-TA request is received from the 1609.3 WME, containing the repeat rate, channel, channel interval, and destination MAC address.

2. Verify that we are on the correct channel and that the interval is set to both CCH and SCH intervals.

3. Setup a timer to provide callbacks according to the repeat rate.

4. When the timer expires, transmit one Timing Advertisement frame.

In this case, we would not have to worry about not being able to transmit a TA frame due to being on the wrong channel or wrong channel interval. All TA frames would be transmitted according to the repeat rate. This case can be seen in Figure 3.5. However, if we are currently using alternating channel access, this algorithm would not work. Since channels are being switched every 50 ms, we may not be on the correct channel at the time that the timer expires. The obvious solution is to wait until we reach the desired channel and then transmit the TA. If this was the case, depending on the repeat rate, we may have multiple TAs waiting to transmit by the time we arrive on the correct channel. In a typical queueing scheme, we would queue TA frames while we are not able to transmit them, and then transmit all of them, one after the other, once we are on the correct channel. If the TA frames are queued, the timing information would no longer be valid by the time they are transmitted. We could keep a count of the number of TAs that are waiting to be sent, and then generate them all at the time that we reach the correct channel, but this would cause a number of Timing Advertisements to be sent containing nearly identical timing information.

Our solution to this is to keep a count of TAs that could not be sent due to being on the wrong channel, and then once we reach the correct channel, space the generation and transmission of the waiting TA frames evenly throughout the channel interval. This way, the correct number of TAs are being sent, and we still have some spacing between TA frames. The algorithm is as follows, and can be seen in Figure 3.6.

1. Upon receiving MLMEX-TA request from WME, set up and start Timer 1 according to the repeat rate received in the request.

2. When Timer 1 fires, check if channel and interval from request match current channel. If channel and interval match, send TA now, else increment count of waiting TAs and wait for correct channel.

3. When we reach the correct channel and interval, calculate period to use for Timer 2 by dividing the length of the channel interval by the number of waiting TAs.

4. Set up and start Timer 2 at the beginning of the correct channel interval.

5. When Timer 2 fires, send a TA frame and decrement the number of waiting TAs.

6. When the channel changes, stop Timer 2.

The MLMEX-TAEND request primitive is used to stop transmission of Timing Advertisements. This is implemented by simply canceling both of the timers.

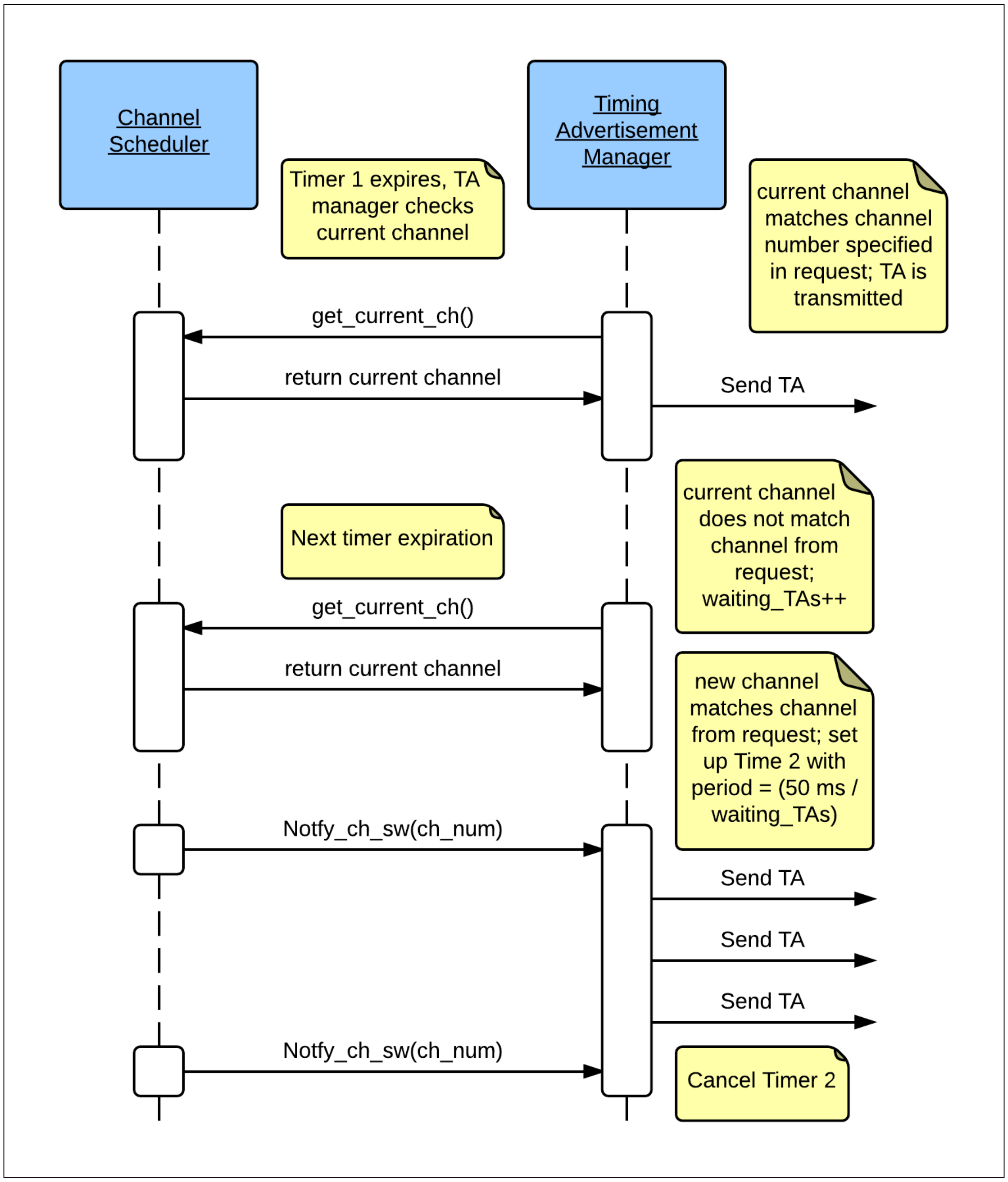

Channel Scheduler
Timing Advertisement Manager
Timer 1 expires, TA manager checks current channel
current channel matches channel number specified in request; TA is transmitted
get_current_ch()
return current channel
Send TA
Next timer expiration
current channel does not match channel from request; waiting_TAs++
get_current_ch()
return current channel
new channel matches channel from request; set up Time 2 with period = (50 ms / waiting_TAs)
Notfy_ch_sw(ch_num)
Send TA
Send TA
Send TA
Notfy_ch_sw(ch_num)
Cancel Timer 2

**Figure 3.6**: Transmission of TAs during alternating channel access

*Structure*

In order to implement the primitives that deal with Timing Advertisements, a *Timing Advertisement Manager* entity was created. The *Timing Advertisement Manager* consists of 2 internal structs and several internal helper functions defined in *time_adv_mgr.c*, and 3 public functions declared in *time_adv_mgr.h*:

1. **void ta_mgr_init(struct ieee16094_ch_sched_t *ch_sched);**

2. **int ta_mgr_start(const u8 *mac_addr, u8 repeat_rate, u8 ch_num, enum wave_ch_interval interval);**

3. **int ta_mgr_end(u8 ch_num);**

The functions *ta_mgr_start()* and *ta_mgr_end()* correspond to the MLMEX-TA primitive and the MLMEX-TAEND primitive, respectively. Howver, prior to calling either of these, *ta_mgr_init()* must first be called. In this 1609.4 implementation, *ta_mgr_init()* is called when the top-level 1609.4 entity is initialized.

These public functions are implemented using 2 structs:

1. **struct ieee16094_ta_mgr_t**

   This represents the *Timing Advertisement Manager* at its highest level, and contains all data needed to implement the public functions.

2. **struct ta_timer**

   This represents one of the two timers used by the *Timing Advertisement Manager*, and contains all data needed to manage these timers.

The struct *ieee16094_ta_mgr_t* contains two instances of *struct ta_timer*, one that provides callbacks at the specified repeat rate, and one that is used to divide a channel interval according to the number of TAs that are waiting to be sent. It contains a pointer to the *Channel Scheduler*, in order to get callbacks on channel switches, as well as information about the current channel and access mode. It also contains a count of the number of waiting TAs, as well as fields to store the values of the parameters received in the MLMEX-TA request.

### *APIs Used*

The timers were implemented using the Linux kernel's high resolution timer APIs [20]. These APIs allow the creation of timers that operate in terms of nanoseconds. A high resolution timer is represented by the struct *hrtimer*. An *hrtimer* is initialized by calling *hrtimer_init()*. This function takes a pointer to an *hrtimer*, a *clock_id* to specify which system clock to use for this timer, and a *mode* - either absolute or relative time. For this implementation, the real-time system clock was used, for consistency with the channel scheduler, and relative time was used, since these timers are to start in response to events, such as receiving an MLMEX-TA request or at the start of a channel interval.

After an *hrtimer* has been initialized, it is started by calling *hrtimer_start()*. In addition to a pointer to the *hrtimer*, this function takes a *time* when it is to expire, and a *mode*. The *time* parameter is of type *ktime_t*, and in order to set the value, the function *ktime_set()* is used. This allows a *ktime_t* to be set given a number of seconds and a number of nanoseconds.

The *hrtimer* struct has a *function* field, which a callback function is assigned to. This callback takes a pointer to the *hrtimer* as a parameter, and must return an *enum hrtimer_restart* value. This value can be either *HRTIMER_RESTART* or *HRTIMER_NORESTART*, indicating that the timer should restart or cancel, respectively. In most timer APIs, if you want the timer to expire periodically, the period is specified prior to starting the timer, and the callback is called at a set interval. However, in the *hrtimer* API, the callback function is responsible for setting the next expiration time for the timer. This can be done using the function *hrtimer_forward()*. The callback functions for both timers used in the *Timing Advertisement Manager* handle calculation and setting of the next expiration time, as well as calling a helper function to either transmit a TA frame, or increment the number of waiting TAs, depending on the current channel and interval.

Another challenge in the implementation of Timing Advertisements in this manner, is that the timer callbacks are called from an interrupt. When we are in interrupt context, functions that may sleep or otherwise take a long time to execute cannot be called [22]. Because of this, the transmission of Timing Advertisement frames cannot be done in the timer callback, but must be taken care of after the callback returns. The Linux kernel provides *workqueues* as a mechanism to defer calls to functions that may sleep [21].

In the Linux kernel, interrupt handlers are divided into two parts, the top-half and the bottom-half [22]. The top-half of an interrupt handler is executed immediately when the interrupt occurs, and the bottom-half is executed at a later time, when a function call may sleep or take a longer time to execute. The reason for this distinction is that during a top-half interrupt handler, other interrupts may be disabled. During a bottom-half interrupt, all other interrupts are enabled, allowing other interrupts to be serviced at the same time that a bottom-half interrupt is being executed. There are two mechanisms that can be used for bottom-half interrupt handling, *tasklets*, and *workqueues*. Tasklets have lower latency, but all operations must be atomic. Since the call to transmit a Timing Advertisement frame may sleep, workqueues are used. Workqueues are not as fast as tasklets, but they may sleep if necessary.

In order to use a workqueue to defer function calls, the structs *workqueue_struct* and *work_struct* are used [21]. A *workqueue_struct* is created using the function *alloc_workqueue()*, taking a parameter for priority and for flags. A *work_struct* is created with a call to *kmalloc()*, just as with any other structure allocated in the kernel. After the structs have been allocated, the *work_struct* is associated with a bottom-half interrupt handler function using the macro *INIT_WORK()*. This handler function takes a pointer to the *work_struct* that it is associated with, and returns void. In the implementation of Timing Advertisements, a handler *send_ta()* was defined in order to transmit a single TA frame and decrement the number of waiting TAs.

In order to schedule work to be transmitted, the function *queue_work()* is used. This function takes pointers to the *workqueue_struct* and the *work_struct*. When this is called, the work contained in the *work_struct*, specifically, the bottom-half interrupt handler function (in our case, *send_ta()*), is enqueued on to the workqueue contained in the *workqueue_struct*. This is done from interrupt context, in the top-half interrupt handler. The deferred work is then executed once we are not in interrupt context anymore. This is done by a kernel worker thread dequeueing the work and executing the bottom-half interrupt handler. Since we are not in interrupt context anymore, these handlers are able to execute concurrently.

In the implementation of the *Timing Advertisement Manager* the use of workqueues to transmit a Timing Advertisement frame is as follows:

1. When the *Timing Advertisement Manager* is initialized, the *workqueue_struct* and *work_struct* are created, and the *work_struct* is associated with the bottom-half interrupt handler, *send_ta()*.

2. When we want to transmit a Timing Advertisement from a timer callback, *queue_work()* is called, adding an instance of the *work_struct* containing *send_ta()* to the workqueue.

3. When the timer callback returns and we are no longer in interrupt context, a kernel worker thread dequeues a *work_struct* from the workqueue and executes the bottom-half handler, *send_ta()*.

### 3.4.3 Management Information Base

The Management Information Base (MIB), contains status and configuration information relating to the 1609.4 entity. The information to be contained in the MIB is specified in Annexes D and E of the standard. This information is separated into MIB items, which contain variables indicating the status and capabilities of the device. The MIB items are as follows:

1. **Capabilities** - This contains information related to features that are supported or not supported for this WAVE device.
2. **Switching** - This contains the lengths of the channel intervals and guard interval components.
3. **ChannelSetTable** - This is a table of channels supported by the device.
4. **EDCACchTable** - This is a table containing values of the EDCA parameters for the control channel.
5. **EDCASchTable** - This is a table containing values of the EDCA parameters for the service channels.
6. **TransmitterProfileTable** - This contains transmitter profiles for service channels that may transmit IP-based data.
7. **TimingInformation** - This contains information relating to the device's time source and UTC synchronization status.

These items are implemented as structs defined in *mib.c*. Instances of these structs are contained in the top-level MIB struct, *ieee16094_mib_t*, which is also defined in *mib.c*. Like other entities in this implementation, the definitions of the MIB structs are private, so that the fields cannot be accessed directly by code external to the MIB. There is a public declaration of the struct *ieee16094_mib_t*, located in *mib.h*.

The MIB is initialized by calling the function *mib_init()*, which returns a pointer to a new *ieee16094_mib_t*. MIB items are accessed and modified by calls to the functions declared in *mib.h*. The MIB can be destroyed by calling *mib_destroy()*.

The majority of the functions declared in *mib.h* are used for accessing or modifying the fields of the *ieee16094_mib_t*. These will not be discussed in-depth, as they are not of high importance to the overall 1609.4 functionality. However, the MIB does contain some functions that implement required primitives for 1609.4:

1. **int mib_reg_tx_profile(struct ieee16094_mib_t *mib, u8 ch_num, bool adaptable_pwr_and_rate, u8 data_rate, u8 pwr_level);**

2. **mib_unreg_tx_profile(struct ieee16094_mib_t *mib, u8 ch_num);**

3. **void mib_get_utc_time(struct ieee16094_mib_t *mib, struct tm *time, __kernel_suseconds_t *usec, u8 time_error[5]);**

4. **int mib_set_utc_time(struct ieee16094_mib_t *mib, const struct tm *time, __kernel_suseconds_t usec, u8 time_error[5]);**

The first two functions, *mib_reg_tx_profile()* and *mib_unreg_tx_profile()*, correspond to the MLMEX-REGISTERTXPROFILE and MLMEX-DELETETXPROFILE primitives, respectively. The functions *mib_get_utc_time()* and mib_set_utc_time() correspond to the MLMEX-GETUTCTIME and MLMEX-SETUTCTIME primtives.

### *Transmitter Profile Table*

The transmitter profile table is used in order to retrieve the channel, data rate, and transmit power level to be used in transmission of IPv6 data, since this data is not included in the MA-UNITDATA request. The standard is vague here, as it does not mention how we know which transmitter profile from the table is to be used with each MA-UNITDATA request received from the LLC sublayer. If the MA-UNITDATA request was modified in order to clarify this, it would break compatibility with existing IPv6 applications. Because of this, our implementation only uses the first transmitter profile from the table, meaning that although we can have up to 6 active service channels, IPv6 data can only be sent on one of them. All other service channels would transmit WSMP data only.

Although we only use the first transmitter profile, the table was implemented as an array of *struct tx_profile_entry_t* instances, with a size of WAVE_NUM_SCHS (defined as 6). This was done so that the underlying structure specified in the standard does exist in this implementation, and its use can be extended to multiple transmitter profiles in the future as ambiguities in the standard are resolved. This struct contains the following fields:

1. **u8 ch_num;**

2. **bool adaptable_pwr_and_rate;**

3. **u8 data_rate;**

4. **u8 pwr_lvl;**

The fields *ch_num*, *data_rate*, and *pwr_lvl* are the transmit parameters to use for the transmission of IP data. The field *adaptable_pwr_and_rate* indicates whether the transmit power level and data rates are adaptable. If true, the fields *data_rate* and *pwr_lvl* are upper bounds, rather than absolute values.

Implementation of *mib_reg_tx_profile()* and *mib_unreg_tx_profile()* was very straight forward. In order to register a transmitter profile, the number of stored transmitter profiles is incremented, and the fields in the last profile are populated according to the parameter values. In order to remove a transmitter profile, the table is searched by channel number. Once the profile is found, then all subsequent entries are shifted down, overwriting the profile. Then the number of profiles is decremented. Although it is usually more efficient to use a linked-list when inserting and removing items, the transmitter profile table has a max size of 6 elements and is not constantly changing, and it is simpler to implement using an array.

### *Timing Information*

The MIB element *TimingInformation* is responsible for keeping the values *TimingCapabilities*, *TimeValue*, and *TimeError*. *TimingCapabilities* indicates the WAVE device's source of external time. From the standard, possible values are:

**0** Indicates no standardized external time source. This is represented by the constant **WLAN_TIMING_NO_STD_EXTERNAL_SOURCE**.

**1** Indicates that the *TimeValue* is an offset based on UTC. This is represented by the constant **WLAN_TIMING_TIMESTAMP_OFFSET_UTC**.

**2** Indicates that *TimeValue* is the UTC time at which the TSF timer is 0. This is represented by the constant **WLAN_TIMING_UTC_TIME**.

The *TimeError* field represents the standard deviation of error of the *TimeValue*. The *TimingInformation* MIB element is represented with the struct *timing_info_t*. This struct contains fields that correspond to the *TimingCapabilities* and the *TimeError*. *TimeValue* is calculated when the MLMEX-GETUTCTIME request is received, by subtracting the value of the TSF timer from the current UTC time.

The function *mib_get_utc_time()* returns the *Timestamp*, *TimeValue*, and *TimeError* by accepting pointers to their corresponding types, and then modifying the contents being pointed to. In addition to these required output parameters, the current system time can be returned in the same way. There is also a parameter for the system time, *struct timeval *sys_time*. This functionality was added since it may be more useful to the upper layer than only providing the offset, since other implemented functionality, such as channel switching, is based on the system time rather than an offset.

The function *mib_set_utc_time()* uses the *TimeValue* element in order to update the system time, and also accepts a value for *TimeError*, which will be stored in the MIB. The *TimeValue* is not stored, as it is calculated as needed. Since the system time is updated, the *TimeValue*, when calculated, should be equivalent to the *TimeValue* passed to this function. Since the system time may be updated directly by the GPS in our implementation, the *TimeValue* may change, which is why it is calculated as needed rather than being stored.

# CHAPTER 4
# TESTING AND VERIFICATION

The management primitives specified in the standard are implemented by the MIB, Channel Scheduler, and Timing Advertisement Manager, and the data plane primitive, MA-UNITDATAX is implemented by modifying the existing implementation of the MA-UNITDATA primitive. Testing of this implementation consists of verifying that these primitives perform according the the IEEE 1609.4 specification. The primitives implemented by the MIB, Channel Scheduler, and Timing Advertisement Manager are all tested individually, as well as being tested together as part of a test of the overall transmit operation. This test also covers the existing MA-UNITDATA primitive as well as the modifications to support the requirements of the MA-UNITDATAX primitive.

This chapter starts by describing Linux's *debugfs* API, as it was used extensively throughout the testing process. Following this are sections for the MIB, Channel Scheduler, and Timing Advertisement Manager entities, and the overall 1609.4 transmit operation. Each of these sections starts with a description of the debugfs entries implemented, then a description of the testing procedure and results.

## 4.1 DEBUGFS

Since the implementation of 1609.4 is a kernel module, testing from userspace presents a challenge: kernel functions cannot be called directly from userspace. In order to overcome this challenge, the Linux kernel provides an API for *debugfs*, a virtual file system that can be used for debugging kernel code [29]. In Linux, *procfs* and *sysfs* can also be used for accessing kernel information from userspace, but they serve specific purposes and have rules governing their usage [30]. *Procfs* is used to provide information about processes, and *sysfs* is used to provide information about devices and drivers, and is very structured, as it represents the entire device hierarchy of the kernel. Debugfs is intended to provide similar functionality, but without the rules and restrictions of *proc* and *sysfs*. Debugfs was used for testing and debugging of all of the 1609.4 entities implemented.

Debugfs is a *virtual file system*. Its files are visible in the file system, and standard file operations can be performed on them, such as reading from and writing to. However, these files do not actually exist on disk. When a file is read from or written to, the *read()* or *write()* operations associated with the virtual file are called. These operations are implemented by the creator of the debugfs entries, and are able to call functions from kernelspace.

Implementation of debugfs entries is done using the following process:

1. Functions for the desired file operations are implemented: *read()*, *write()*, or both. The implementations of these can call any kernel functions necessary. Any data passed in at the creation of the file can be accessed using the *private_data* field of the file structure.

2. An instance of the struct *file_operations* is declared, and the function pointers are initialized to point to the corresponding implemented operations.

3. A debugfs entry is then created using the function *debugfs_create_file()*. Among the parameters accepted by this function are *const struct *file_operations* and *void *data*. The *file_operations* pointer passed should contain the address of the *file_operations* struct containing the operations to be associated with the file. The argument passed to the *data* parameter can point to any data needed for the implementation of the file operations.

There are also debugfs functions available to create files for reading and writing to a single value, so that the file operations do not have to be implemented. However, these functions were not used in our case, since the data being read or written was more complex than simple variable assignment.

In order to access the debugfs entries, the debugfs file system must be mounted at */sys/kernel/debug/*, using the command *mount -t debugfs none /sys/kernel/debug/* [31]. The *mount -t* command takes the arguments *type*, *device*, and *directory*. In the case of debugfs, the type is *debugfs* and there is no device (since the files are virtual and do not exist on disk), so the device is *none*. Within the top-level directory are subdirectories (these can be created with the *debugfs_create_dir()* function) and debugfs files.

A file can be written to and read from just like any files that are located on disk, using standard command line programs like *echo* and *cat*, or by using the file input / output utilities of a programming language. For our purposes, the *read()* function for a debugfs entry was used to print status information and values of variables, and the *write()* function was used to either change the values of variables, or to initiate some 1609.4 functionality.

For example, there is a debugfs entry, *sch_start*, that corresponds to the MLMEX-SCHSTART primitive. This can be used to start providing service channel access, for example, on channel 172, with the following command: *echo channel=172 immediate=0 extended=0 >sch_start*. The *write()* function associated with the *sch_start* file parses the arguments and then calls the *ch_sched_sch_start()* function, passing the appropriate arguments. As an example for reading from a debugfs file, we can check the status of the channel scheduler by reading from the channel scheduler's *status* file, using the command: *cat status*. In this case, the *read()* function associated with the *status* file calls internal functions to get the channel scheduler's access mode and current channel, then a string is returned containing this information.

## 4.2 MANAGEMENT INFORMATION BASE

Although the majority of the 1609.4 entities depend on the MIB, the MIB itself does not depend on any other parts of 1609.4. Because of this, the testing was very straightforward. Since the MIB's main responsibility is to store data, most of the functionality consists of accessors and mutators for the data items contained in the MIB. This means that the majority of the testing consists of:

1. Setting the value of some data stored in an MIB item using a mutator function.
2. Retrieving the new value of the same data item using an accessor function.
3. Verifying that the accessed data matches the expected value.

In order to do this, a test function, *mib_test()* was implemented as part of the MIB. This is called after the MIB has been initialized. This function prints the contents of the MIB to the kernel log in order to verify that the MIB contains the correct default values, and goes through a sequence of function calls from the MIB, checking the return values and printing the contents of the corresponding MIB items before and after the calls. The kernel log contents can be viewed using the *dmesg* command.

The remainder of this section will describe the debugfs entries for the MIB, and testing of the primitives that are implemented by the MIB entity.

### 4.2.1 Debugfs

A debugfs directory, *mib*, was created in the *ieee16094* debugfs top-level directory, in order to allow MIB data to be accessed and functions to be called during testing. The contents of this directory are as follows:

**capabilities** This is a read-only file that prints out the contents of the MIB capabilities item.

**channel_set_table** This is a directory that contains a file for each supported channel. Each of the files contained within this directory is read-only, and prints the values of the fields specified in the channel set table.

**edca_cch_table** This is a directory containing a file for each of the 4 access categories. Each of these access catefory files is read only and prints out the contents of the EDCA parameters for the corresponding access category on the control channel.

**edca_sch_table** This is the same as above, but for service channel EDCA parameters.

**switching_capabilities** This is a read-only file that prints the values of the switching capabilities MIB item.

**timing_capabilities** This file allows both read and write operations, and represents the value of the timing capabilities MIB item.

**tx_profile_table** This directory represents the transmitter profile table. It contains a file, *reg_tx_profile*, which a channel number can be written to in order to register a new transmitter profile. Once a transmitter is registered, a debugfs file named after the channel is created. This transmitter profile can then be modified or deleting by writing appropriate arguments to the file.

### 4.2.2 Testing Procedure and Results

In addition to the *mib_test()* function for testing the contents of the MIB, testing also consisted of verifying the proper behavior of the following primitives implemented by the MIB:

1. MLMEX-REGISTERTXPROFILE
2. MLMEX-DELETETXPROFILE
3. MLMEX-GETUTCTIME

4. MLMEX-SETUTCTIME

The first two of these primitives are used for managing the MIB's transmitter profile table, and the last two are used for getting and setting the device's UTC time estimate.

*Transmitter Profile Table*

The transmitter profile table functions were tested extensively in the *mib_test()* function. This contains a sequence of function calls to register and unregister transmitter profiles. The contents of the transmitter profile table are printed to the kernel log before and after registering or unregistering profiles, and the return values are checked for cases when the table is full, a transmitter is registered multiple times, or when invalid parameters are passed. The debugfs entries for the transmitter profile table are also used for manual testing from the command line.

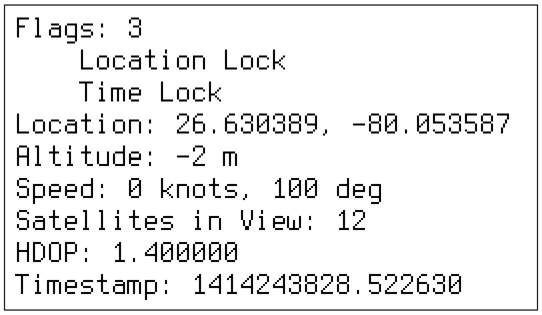
*UTC Timing*

The MLMEX-GETUTCTIME and MLMEX-SETUTCTIME primitives are implemented in the functions *mib_get_utc_time()* and *mib_set_utc_time()*. As described in Chapter 3, the standard specifies the time value as an offset from the TSF timer. This offset is returned as *time_value* from *mib_get_utc_time()* along with *timestamp*, the value of the TSF timer itself. The function *mib_set_utc_time* takes the offset, *time_value*, and uses it, along with the TSF timer value obtained from *mac80211*, to calculate and set the UTC time.

Testing of these functions involves making sure that *mib_set_utc_time()* sets the UTC time correctly according to the *time_value*, and making sure that *mib_get_utc_time()* returns the correct *time_value*. This testing was done in two separate procedures, one to verify that the UTC time is updated correctly, and one to verify that the returned *time_value* is correct. For verifying that the UTC time is updated correctly, a test was done involving 2 devices and the transmission of Timing Advertisements. The procedure is as follows:

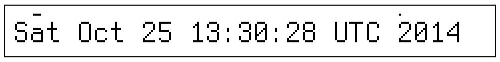
1. Device 1 is synchronized to UTC time using GPS, device 2 is not synchronized.
2. Both devices are configured for continuous control channel access.
3. Device 1 begins broadcast transmission of Timing Advertisements.
4. Upon receipt of a Timing Advertisement, device 2 updates its time using *mib_set_utc_time()*.

By displaying the system time, synchronization can be verified only down to the level of seconds. However, it is safe to assume that if the system time was set correctly for year, month, day, hour, minute, and second, that the microseconds were also set correctly, as this is all done using an existing Linux kernel function. This assumption, however, is only valid for the actual setting of the UTC time, not for synchronization between devices, as there can be some error due to transmission time. Synchronization is tested separately with Timing Advertisements. Output of this test is shown in Figures 4.1 through 4.4. Note that the times shown in Figures 4.2 and 4.4 are now the same, with the exception of seconds. This difference is due to time that passed while typing in the commands.



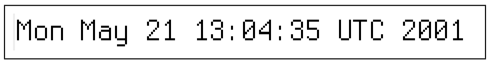
```
Flags: 3
    Location Lock
    Time Lock
Location: 26.630389, -80.053587
Altitude: -2 m
Speed: 0 knots, 100 deg
Satellites in View: 12
HDOP: 1.400000
Timestamp: 1414243828.522630
```

**Figure 4.1**: GPS information for the transmitting device, showing that we have a time lock



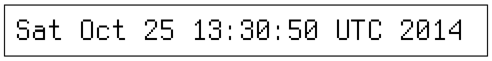
```
Sat Oct 25 13:30:28 UTC 2014
```

**Figure 4.2**: UTC system time on transmitting device prior to sending a Timing Advertisement



```
Mon May 21 13:04:35 UTC 2001
```

**Figure 4.3**: UTC system time on receiving device prior to the receipt of a Timing Advertisement



```
Sat Oct 25 13:30:50 UTC 2014
```

**Figure 4.4**: UTC system time on receiving device after the receipt of a Timing Advertisement

Since the *time_value* is used by *mib_set_utc_time()* to set the system time, and it is also calculated in *mib_get_utc_time()*, the *time_value* returned from *mib_get_utc_time()* should be equal to the *time_value* that was passed to *mib_set_utc_time()*. In order to verify this, testing was done within the *mib_test()* function, using the following procedure:

1. The function *mib_set_utc_time()* is called, passing a *time_value*. This *time_value* is used to set the system time, but it is not stored.

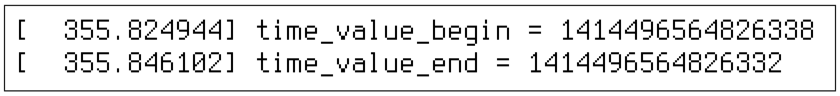
2. Now, the system time has been updated, so that the current time = TSF + *time_value*.

3. The function *mib_get_utc_time()* is called, returning a newly calculated *time_value*. This is calculated by subtracting the TSF value from the current

time. Since the system time and the TSF update at the same rate, the first and second *time_values* should be equal.

4. Since the TSF and system time update in microseconds, there still may be a variance in the first and second *time_values* due to processing time. The values are printed to the kernel log, and the difference is checked to ensure that it is acceptable.

This test was run several times, and the differences between the first and second *time_values* were always less than 20 microseconds, typically between 2 and 10. According to the standard, there is a *SyncTolerance* of 2 milliseconds for channel switching, and a maximum of 20 microseconds falls well below this. The output of a single test run is shown in Figure 4.5.



```
[  355.824944] time_value_begin = 1414496564826338
[  355.846102] time_value_end = 1414496564826332
```

**Figure 4.5**: Output of test for *time_value* equality

## 4.3 CHANNEL SCHEDULER

Testing of the Channel Scheduler was more involved than that of the MIB, as its functionality is more complex, and we needed to verify that channel switching was happening at the right times, down to the milliseconds, and on the right channels. Due to this complexity, it was not feasible to run a test function at the Channel Scheduler's initialization. Like the MIB, debugfs entries were created and used to test the Channel Scheduler, but rather than just getting and setting data, they initiate channel switching functionality.

### 4.3.1 Debugfs

A directory for the Channel Scheduler, *channel_access*, was created within the *ieee16094* debugfs directory. Within this directory, are 4 debugfs files:

**log** This file is write-only, and is used to start and stop logging channel switches to the kernel log.

**sch_start** This file is also write-only, and invokes the MLMEX-SCHSTART primitive.

**sch_end** This file is write-only, and invokes the MLMEX-SCHEND primitive.

**status** This file is read-only, and prints out the current channel access mode, as well as the active channel.

### 4.3.2 Logging Channel Switches

An important part of testing the Channel Scheduler is to verify that the channel switches happen at the expected time, and that the expected channel is switched to. In order to do this, a message is printed to the kernel log every time a channel switch happens. This message contains the time that the channel was switched, the channel that was switched to, and the channel interval that we are in. This functionality did need to be separate from the Channel Scheduler itself, so it was implemented in the file *ch_sched_debug.c* along with the debugfs entries. The functionality implemented in this file will be referred to as the *Channel Scheduler Debug* entity.

The Channel Scheduler Debug entity receives a notification from the Channel Scheduler every time that the channel was switched, through the use of a function pointer. As described in 3, the Channel Scheduler uses the *observer* pattern in order to notify the Timing Advertisement Manager of channel switches. The same is done here: the Channel Scheduler Debug entity registers a function with the Channel

Scheduler to be called every time that the channel is switched. In addition to receiving these notifications when the channel switches, notifications are also received when the MLMEX-SCHSTART and MLMEX-SCHEND primitives are called.

This debug function called when channels are switched, prints a message to the kernel log to be viewed with the *dmesg* command. There is an issue with this, however. We don't want to constantly flood the kernel log with these channel switch messages. Any other kernel log messages will get lost when we print a channel switch message every 50 ms. Because of this, the Channel Scheduler Debug contains a global boolean variable, *started*, that is set to true when *ch_sched_sch_start()* is called. If *started* is true, then a count of channel switches is maintained, and messages are only printed to the log for the first 20 channel switches. This is usually enough to verify that channel switching is operating correctly, but there are some cases that we need to see what is happening before the MLMEX-SCHSTART request (in the case that alternating service channel access is already being provided), or see more than 20 channel switches. This is where the *log* debugfs entry is used.

The *log* debugfs entry starts or stops logging of channel switches. When "start" is written to the file, a global variable, *logging*, is set to true. When this is true, all channel switch messages are printed to the kernel log. In order to stop printing the messages, "stop" can be written to the *log* file, which sets *logging* to false.

### 4.3.3 Testing Procedure

After implementing the debugging functionality, testing of the MLMEX-SCHSTART and MLMEX-SCHEND primitives was conducted for all channel access modes, including multiple concurrent service channel access. To recap, the *ch_sched_sch_start()* function's parameters *extended* and *immediate* determine the channel access mode to be used. The possible values of these parameters and their corresponding access modes are as follows:

1. $immediate\_access$ = false, $extended\_access = 0$

   access_mode = CH_ACCESS_ALTERNATING

2. $immediate\_access$ = false, $1 \leq extended\_access < 255$

   access_mode = CH_ACCESS_EXTENDED

3. $immediate\_access$ = false, $extended\_access = 255$

   access_mode = CH_ACCESS_CONTINUOUS

4. $immediate\_access$ = true, $extended\_access = 0$

   access_mode = CH_ACCESS_IMMEDIATE

5. $immediate\_access$ = true, $1 \leq extended\_access < 255$

   access_mode = CH_ACCESS_EXTENDED

6. $immediate\_access$ = true, $extended\_access = 255$

   access_mode = CH_ACCESS_CONTINUOUS

For each of these cases, another thing to consider is the fact that the MLMEX-SCHSTART request can be received during a CCH interval or an SCH interval. When this is received during a CCH interval, the intended behavior is explicitly specified in the standard by timing diagrams. However, there is no mention of the possibility of the MLMEX-SCHSTART request occurring during an SCH interval. For this implementation, it was decided to treat an interrupted SCH interval in the same way an interrupted CCH interval is handled. If *immediate* = false, we wait until the beginning of the next SCH interval to start switching channels. If *immediate* = true, we switch immediately, interrupting the SCH interval, and return to alternating access when the next CCH interval is reached. This means that all 6 cases above must be tested for when the MLMEX-SCHSTART request is received during a CCH interval and when it is received during an SCH interval.

Since the functionality for starting channel access, and for logging channel switches is implemented using debugfs, the actual testing procedure is fairly simple. For each of the 12 cases:

1. Check the current channel scheduler's status by reading from the *status* file. By default, we should have continuous access on the control channel (178).
2. Start service channel access by writing to the *sch_start* file, passing appropriate arguments for channel number, immediate, and extended access. In addition to starting SCH access, this also logs the first 20 channel switches.
3. Check the kernel log for the channel switch messages. Verify that the channel switches take place at the appropriate time based on the access mode, and the time that the MLMEX-SCHSTART request is received.

In addition to the above testing for the standard channel access modes, multiple concurrent service channel access is tested in the following way:

1. Start with alternating channel access, this is done by writing to *sch_start* with $immediate = 0$ and $extended = 0$, from the default mode.
2. Start continuous logging by writing "start" to the *log* file.
3. Initiate a second MLMEX-SCHSTART primitive, by repeating the first step.
4. Stop continuous logging by writing "stop" to the *log* file.
5. Check the kernel log and verify that we are alternating between the CCH, the first SCH, and the second SCH as expected, with switches happening at the correct times. We will also see that, prior to the second MLMEX-SCHSTART request, we are alternating with only one SCH.
6. Access for up to all 6 service channels can be added by repeating the third step multiple times.

This covers all of the cases tested for the MLMEX-SCHSTART primitive. For the MLMEX-SCHEND primitive, we only have 2 cases to test for: when we have single SCH access, and when we have multiple SCH access. For single SCH access, we just have to verify that channel access on the specified SCH is stopped, and that we return to continuous access on the CCH. For multiple SCH access, we have to verify that once we end SCH access for a single SCH, that we are no longer switching to this SCH, but that we still switch among the remaining SCHs at the correct times. The test procedure for the MLMEX-SCHEND primitive is as follows:

1. Check that we are providing access to one or more service channels by reading from the *status* file.

2. Start continuous logging as described in the previous scenario.

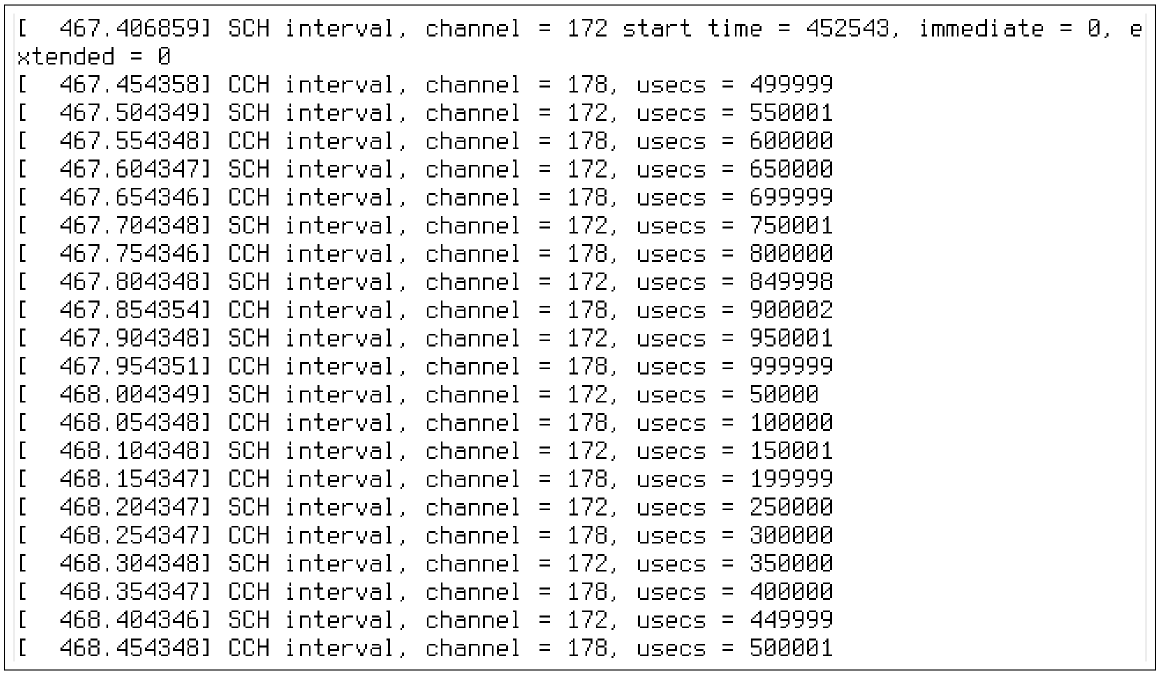
3. End service channel access by writing the channel number to the file *sch_end*.

4. Stop continuous logging.

5. Check the kernel log and verify that upon receipt of the MLMEX-SCHEND primitive that we are no longer switching to the specified channel.

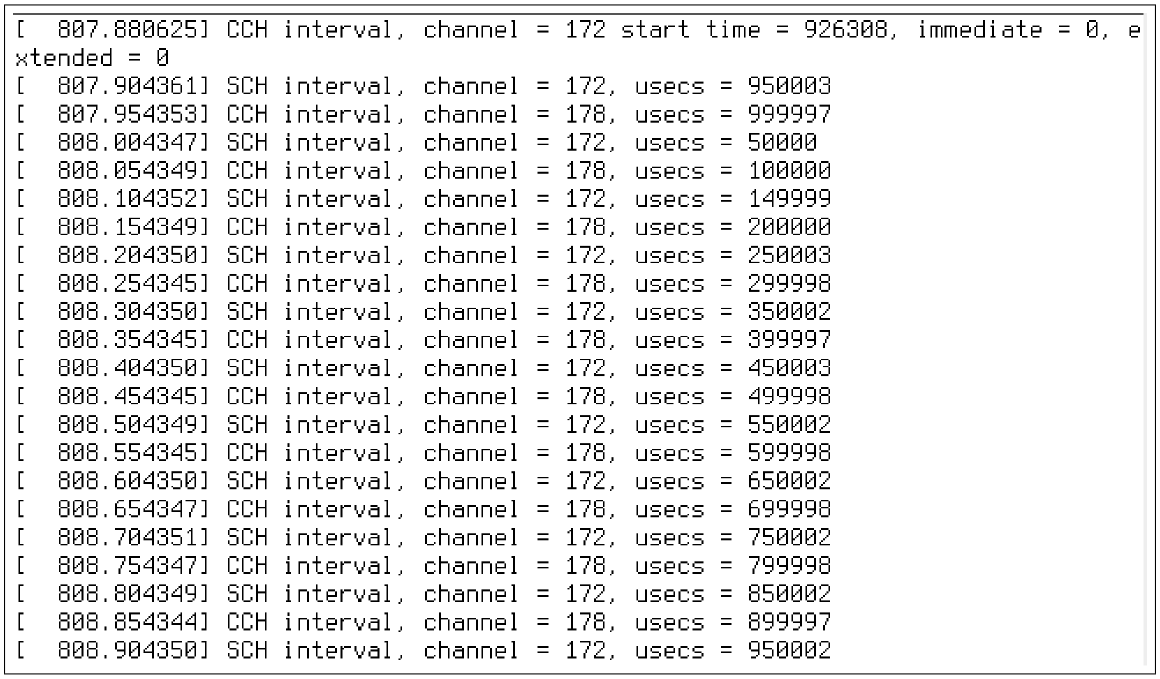
### 4.3.4 Results

The output from the Channel Scheduler tests are shown in Figures 4.6 through 4.25. In these tests, the line containing the start time shows when the request was received, the interval it was received during, the channel to start providing access to, and the values of the *immediate* and *extended* parameters. The following lines show the current interval, channel, and microseconds value of the system time. Channel 178 is the control channel, and channel 172 is used as the service channel in these tests.



```
[  467.406859] SCH interval, channel = 172 start time = 452543, immediate = 0, e
xtended = 0
[  467.454358] CCH interval, channel = 178, usecs = 499999
[  467.504349] SCH interval, channel = 172, usecs = 550001
[  467.554348] CCH interval, channel = 178, usecs = 600000
[  467.604347] SCH interval, channel = 172, usecs = 650000
[  467.654346] CCH interval, channel = 178, usecs = 699999
[  467.704348] SCH interval, channel = 172, usecs = 750001
[  467.754346] CCH interval, channel = 178, usecs = 800000
[  467.804348] SCH interval, channel = 172, usecs = 849998
[  467.854354] CCH interval, channel = 178, usecs = 900002
[  467.904348] SCH interval, channel = 172, usecs = 950001
[  467.954351] CCH interval, channel = 178, usecs = 999999
[  468.004349] SCH interval, channel = 172, usecs = 50000
[  468.054348] CCH interval, channel = 178, usecs = 100000
[  468.104348] SCH interval, channel = 172, usecs = 150001
[  468.154347] CCH interval, channel = 178, usecs = 199999
[  468.204347] SCH interval, channel = 172, usecs = 250000
[  468.254347] CCH interval, channel = 178, usecs = 300000
[  468.304348] SCH interval, channel = 172, usecs = 350000
[  468.354347] CCH interval, channel = 178, usecs = 400000
[  468.404346] SCH interval, channel = 172, usecs = 449999
[  468.454348] CCH interval, channel = 178, usecs = 500001
```

**Figure 4.6**: Alternating access: MLMEX-SCHSTART received during SCH interval



```
[  807.880625] CCH interval, channel = 172 start time = 926308, immediate = 0, e
xtended = 0
[  807.904361] SCH interval, channel = 172, usecs = 950003
[  807.954353] CCH interval, channel = 178, usecs = 999997
[  808.004347] SCH interval, channel = 172, usecs = 50000
[  808.054349] CCH interval, channel = 178, usecs = 100000
[  808.104352] SCH interval, channel = 172, usecs = 149999
[  808.154349] CCH interval, channel = 178, usecs = 200000
[  808.204350] SCH interval, channel = 172, usecs = 250003
[  808.254345] CCH interval, channel = 178, usecs = 299998
[  808.304350] SCH interval, channel = 172, usecs = 350002
[  808.354345] CCH interval, channel = 178, usecs = 399997
[  808.404350] SCH interval, channel = 172, usecs = 450003
[  808.454345] CCH interval, channel = 178, usecs = 499998
[  808.504349] SCH interval, channel = 172, usecs = 550002
[  808.554345] CCH interval, channel = 178, usecs = 599998
[  808.604350] SCH interval, channel = 172, usecs = 650002
[  808.654347] CCH interval, channel = 178, usecs = 699998
[  808.704351] SCH interval, channel = 172, usecs = 750002
[  808.754347] CCH interval, channel = 178, usecs = 799998
[  808.804349] SCH interval, channel = 172, usecs = 850002
[  808.854344] CCH interval, channel = 178, usecs = 899997
[  808.904350] SCH interval, channel = 172, usecs = 950002
```

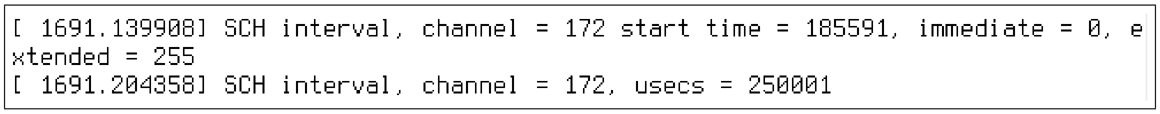
**Figure 4.7**: Alternating access: MLMEX-SCHSTART received during CCH interval

Figures 4.6 and 4.7 show the output of the tests for alternating channel access. In Figure 4.6, the MLMEX-SCHSTART request is received at about 452 ms, during an SCH interval. The next channel interval is the CCH interval at about 500 ms. We can see that at this channel interval, we are on the control channel (178). At the next SCH interval (550 ms), we switch to the service channel (172). For all subsequent intervals, we are alternating between the service channel and the control channel at the SCH and CCH intervals, respectively.

Figure 4.7 shows the same MLMEX-SCHSTART request as in Figure 4.6, but with the request received during a CCH interval. The request in this case is received at about 926 ms. The next channel interval here is the SCH interval at 950 ms, so we switch to channel 172. In all remaining intervals, like the previous case, we alternate between the service and control channels at the SCH and CCH intervals.

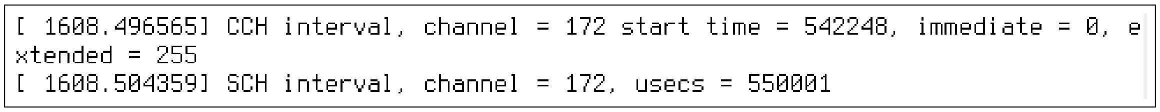
Figures 4.8 and 4.9 show the test output for continuous channel access. In Figure 4.8, the MLMEX-SCHSTART request is received during the SCH interval, at about 186 ms. We do not switch channels at the CCH interval (200 ms), we wait until we reach the next SCH interval at 250 ms. Here we switch to the service channel (172). After this there are no more channel switches.

In Figure 4.9, we show the same case, except that the request is received during a CCH interval, at about 542 ms. In this case, the next channel interval is an SCH interval at 550 ms. Here we switch to channel 172, then there are no more channel switches.



```
[ 1691.139908] SCH interval, channel = 172 start time = 185591, immediate = 0, e
xtended = 255
[ 1691.204358] SCH interval, channel = 172, usecs = 250001
```

**Figure 4.8**: Continuous access: MLMEX-SCHSTART received during SCH interval



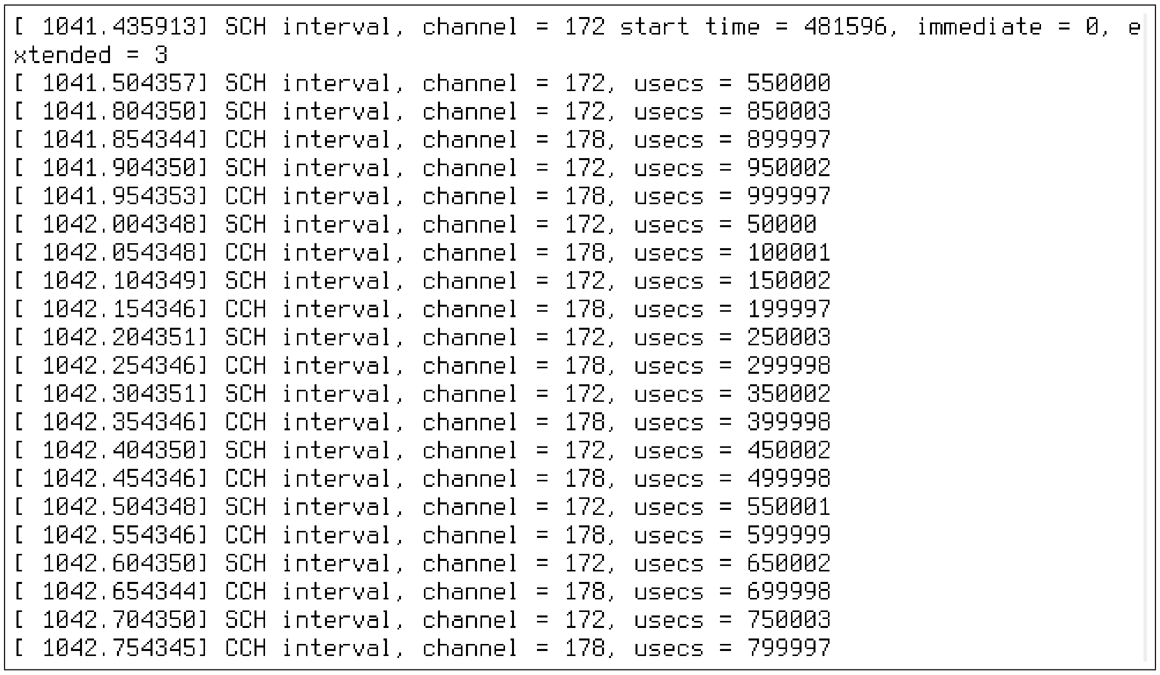
```
[ 1608.496565] CCH interval, channel = 172 start time = 542248, immediate = 0, e
xtended = 255
[ 1608.504359] SCH interval, channel = 172, usecs = 550001
```

**Figure 4.9**: Continuous access: MLMEX-SCHSTART received during CCH interval

Figures 4.10 and 4.11 show when a MLMEX-SCHSTART request is received with *extended* = 3. The value of *extended* indicates the number of CCH intervals to stay on the service channel for, extending access to the service channel beyond a single SCH interval. Since we have a value of 3, this means that once we switch to the service channel, we stay there until after 3 CCH intervals have passed, then we return to alternating access.

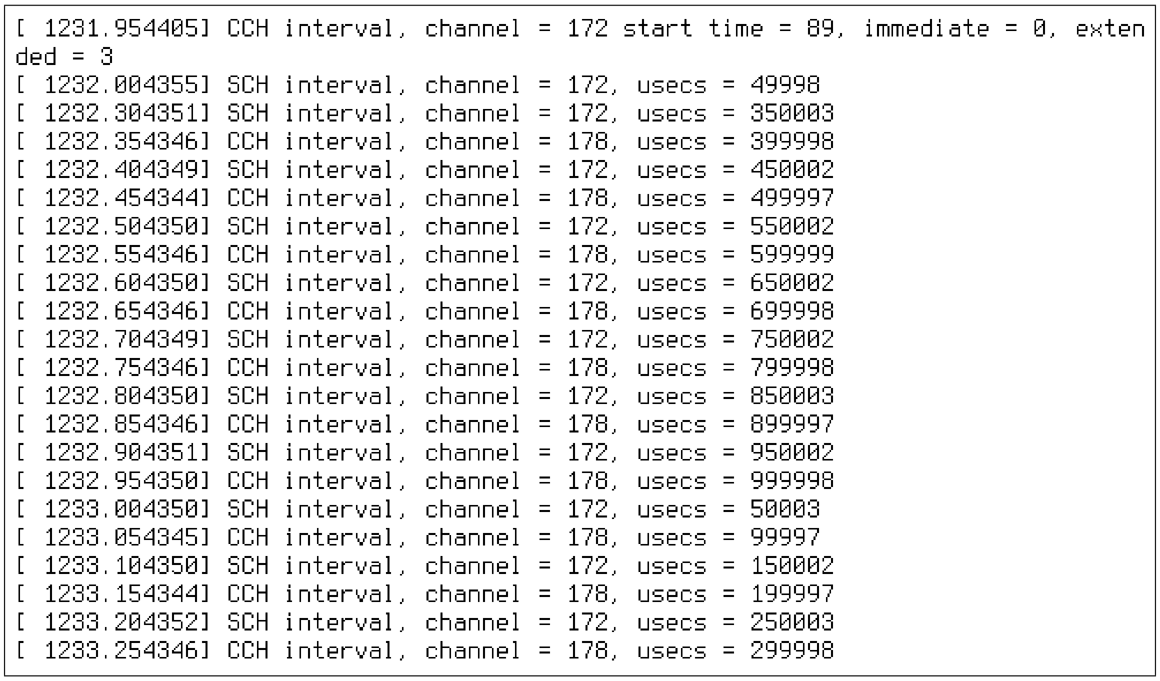
In Figure 4.10, the request is received at 482 ms, during a SCH interval. We wait for the next SCH interval, at 550 ms, to switch to channel 172. We do not switch back to the control channel at the CCH intervals at 600 ms, 700 ms, or 800 ms. After we reach 850 ms, we return to alternating access, switching back to the control channel at 900 ms.

In Figure 4.11, the MLMEX-SCHSTART request is received at about 0 ms, during a CCH interval. The channel switch to 172 happens at 50 ms, the next SCH interval. Now we wait for three CCH intervals to pass, 100 ms, 200 ms, and 300 ms. At 350 ms, we return to alternating channel access and switch back to the control channel.



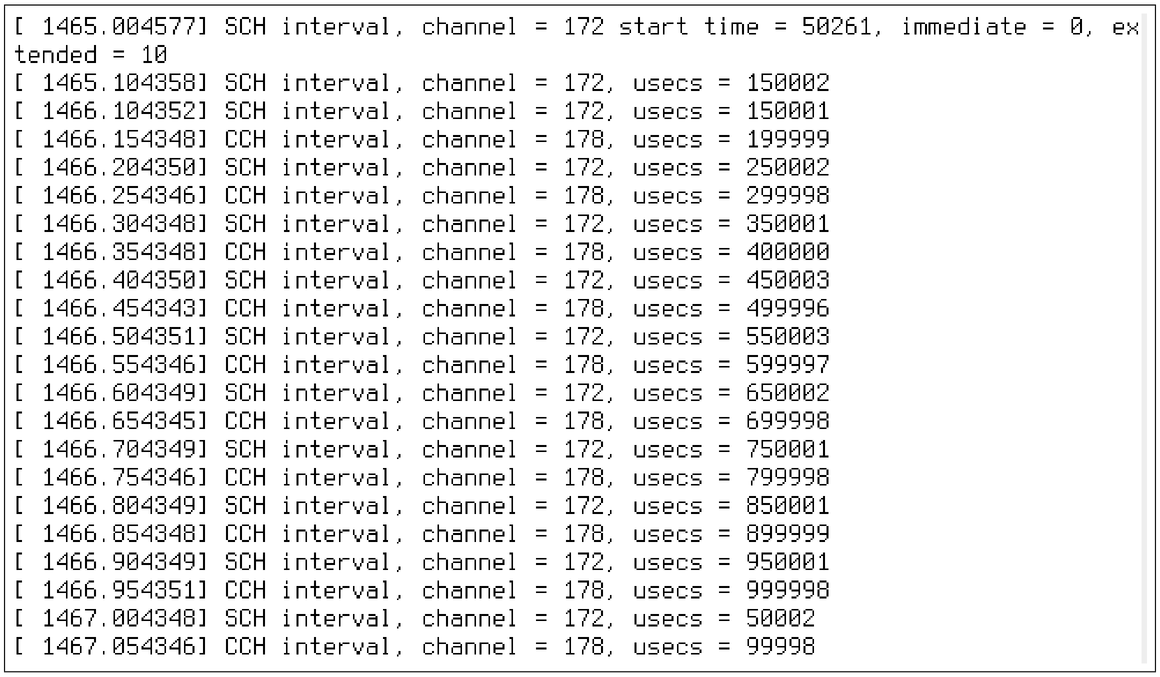
```
[ 1041.435913] SCH interval, channel = 172 start time = 481596, immediate = 0, e
xtended = 3
[ 1041.504357] SCH interval, channel = 172, usecs = 550000
[ 1041.804350] SCH interval, channel = 172, usecs = 850003
[ 1041.854344] CCH interval, channel = 178, usecs = 899997
[ 1041.904350] SCH interval, channel = 172, usecs = 950002
[ 1041.954353] CCH interval, channel = 178, usecs = 999997
[ 1042.004348] SCH interval, channel = 172, usecs = 50000
[ 1042.054348] CCH interval, channel = 178, usecs = 100001
[ 1042.104349] SCH interval, channel = 172, usecs = 150002
[ 1042.154346] CCH interval, channel = 178, usecs = 199997
[ 1042.204351] SCH interval, channel = 172, usecs = 250003
[ 1042.254346] CCH interval, channel = 178, usecs = 299998
[ 1042.304351] SCH interval, channel = 172, usecs = 350002
[ 1042.354346] CCH interval, channel = 178, usecs = 399998
[ 1042.404350] SCH interval, channel = 172, usecs = 450002
[ 1042.454346] CCH interval, channel = 178, usecs = 499998
[ 1042.504348] SCH interval, channel = 172, usecs = 550001
[ 1042.554346] CCH interval, channel = 178, usecs = 599999
[ 1042.604350] SCH interval, channel = 172, usecs = 650002
[ 1042.654344] CCH interval, channel = 178, usecs = 699998
[ 1042.704350] SCH interval, channel = 172, usecs = 750003
[ 1042.754345] CCH interval, channel = 178, usecs = 799997
```

**Figure 4.10**: Extended access = 3: MLMEX-SCHSTART received during SCH interval



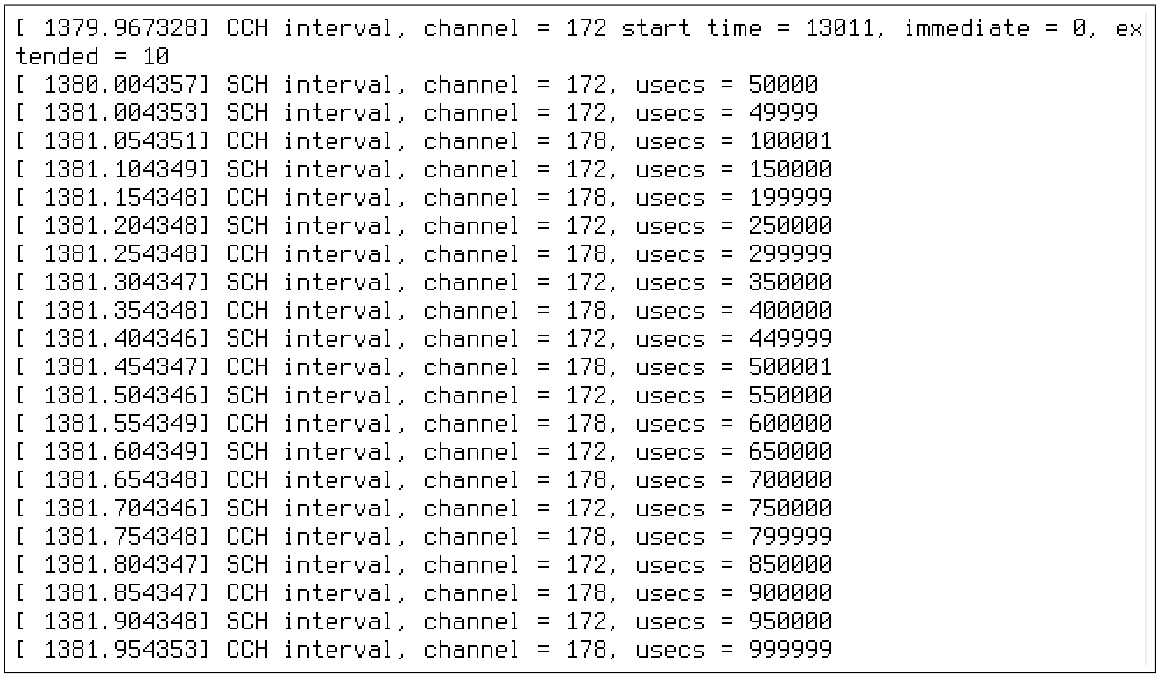
```
[ 1231.954405] CCH interval, channel = 172 start time = 89, immediate = 0, exten
ded = 3
[ 1232.004355] SCH interval, channel = 172, usecs = 49998
[ 1232.304351] SCH interval, channel = 172, usecs = 350003
[ 1232.354346] CCH interval, channel = 178, usecs = 399998
[ 1232.404349] SCH interval, channel = 172, usecs = 450002
[ 1232.454344] CCH interval, channel = 178, usecs = 499997
[ 1232.504350] SCH interval, channel = 172, usecs = 550002
[ 1232.554346] CCH interval, channel = 178, usecs = 599999
[ 1232.604350] SCH interval, channel = 172, usecs = 650002
[ 1232.654346] CCH interval, channel = 178, usecs = 699998
[ 1232.704349] SCH interval, channel = 172, usecs = 750002
[ 1232.754346] CCH interval, channel = 178, usecs = 799998
[ 1232.804350] SCH interval, channel = 172, usecs = 850003
[ 1232.854346] CCH interval, channel = 178, usecs = 899997
[ 1232.904351] SCH interval, channel = 172, usecs = 950002
[ 1232.954350] CCH interval, channel = 178, usecs = 999998
[ 1233.004350] SCH interval, channel = 172, usecs = 50003
[ 1233.054345] CCH interval, channel = 178, usecs = 99997
[ 1233.104350] SCH interval, channel = 172, usecs = 150002
[ 1233.154344] CCH interval, channel = 178, usecs = 199997
[ 1233.204352] SCH interval, channel = 172, usecs = 250003
[ 1233.254346] CCH interval, channel = 178, usecs = 299998
```

**Figure 4.11**: Extended access = 3: MLMEX-SCHSTART received during CCH interval



```
[ 1465.004577] SCH interval, channel = 172 start time = 50261, immediate = 0, ex
tended = 10
[ 1465.104358] SCH interval, channel = 172, usecs = 150002
[ 1466.104352] SCH interval, channel = 172, usecs = 150001
[ 1466.154348] CCH interval, channel = 178, usecs = 199999
[ 1466.204350] SCH interval, channel = 172, usecs = 250002
[ 1466.254346] CCH interval, channel = 178, usecs = 299998
[ 1466.304348] SCH interval, channel = 172, usecs = 350001
[ 1466.354348] CCH interval, channel = 178, usecs = 400000
[ 1466.404350] SCH interval, channel = 172, usecs = 450003
[ 1466.454343] CCH interval, channel = 178, usecs = 499996
[ 1466.504351] SCH interval, channel = 172, usecs = 550003
[ 1466.554346] CCH interval, channel = 178, usecs = 599997
[ 1466.604349] SCH interval, channel = 172, usecs = 650002
[ 1466.654345] CCH interval, channel = 178, usecs = 699998
[ 1466.704349] SCH interval, channel = 172, usecs = 750001
[ 1466.754346] CCH interval, channel = 178, usecs = 799998
[ 1466.804349] SCH interval, channel = 172, usecs = 850001
[ 1466.854348] CCH interval, channel = 178, usecs = 899999
[ 1466.904349] SCH interval, channel = 172, usecs = 950001
[ 1466.954351] CCH interval, channel = 178, usecs = 999998
[ 1467.004348] SCH interval, channel = 172, usecs = 50002
[ 1467.054346] CCH interval, channel = 178, usecs = 99998
```

**Figure 4.12**: Extended access = 10: MLMEX-SCHSTART received during SCH interval



```
[ 1379.967328] CCH interval, channel = 172 start time = 13011, immediate = 0, ex
tended = 10
[ 1380.004357] SCH interval, channel = 172, usecs = 50000
[ 1381.004353] SCH interval, channel = 172, usecs = 49999
[ 1381.054351] CCH interval, channel = 178, usecs = 100001
[ 1381.104349] SCH interval, channel = 172, usecs = 150000
[ 1381.154348] CCH interval, channel = 178, usecs = 199999
[ 1381.204348] SCH interval, channel = 172, usecs = 250000
[ 1381.254348] CCH interval, channel = 178, usecs = 299999
[ 1381.304347] SCH interval, channel = 172, usecs = 350000
[ 1381.354348] CCH interval, channel = 178, usecs = 400000
[ 1381.404346] SCH interval, channel = 172, usecs = 449999
[ 1381.454347] CCH interval, channel = 178, usecs = 500001
[ 1381.504346] SCH interval, channel = 172, usecs = 550000
[ 1381.554349] CCH interval, channel = 178, usecs = 600000
[ 1381.604349] SCH interval, channel = 172, usecs = 650000
[ 1381.654348] CCH interval, channel = 178, usecs = 700000
[ 1381.704346] SCH interval, channel = 172, usecs = 750000
[ 1381.754348] CCH interval, channel = 178, usecs = 799999
[ 1381.804347] SCH interval, channel = 172, usecs = 850000
[ 1381.854347] CCH interval, channel = 178, usecs = 900000
[ 1381.904348] SCH interval, channel = 172, usecs = 950000
[ 1381.954353] CCH interval, channel = 178, usecs = 999999
```

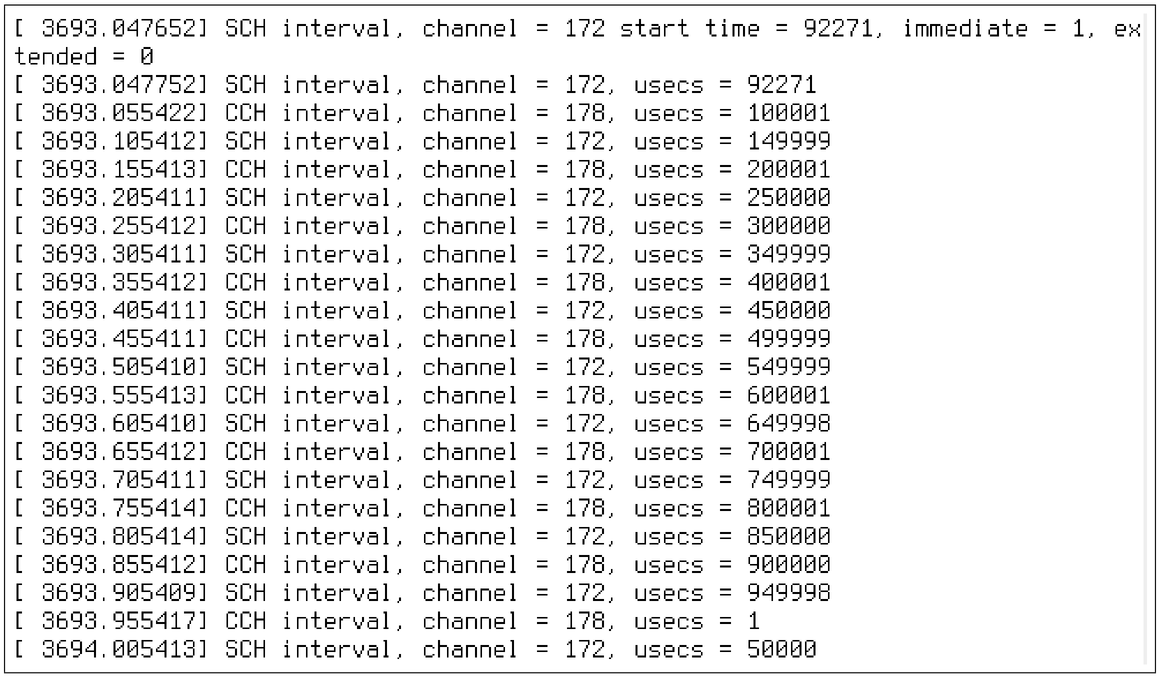
**Figure 4.13**: Extended access = 10: MLMEX-SCHSTART received during CCH interval

The Figures 4.12 and 4.13 show the same scenarios as in Figures 4.10 and 4.11, but with an *extended* value of 10. This means that rather than staying on the SCH for 3 CCH intervals, we stay for 10. In Figure 4.12, the MLMEX-SCHSTART request is received at about 50 ms, during an SCH interval. Our next SCH interval is at 150 ms. Here we switch to channel 172. Now, since the *extended* value is 10, we wait for 10 CCH intervals before returning to alternating access. Since there are 10 CCH intervals in every second, we wait 1 second to return to alternating access. We can see that we return to alternating access at 150 ms past the next second, and we switch to the control channel at 200 ms.

Figure 4.13 shows the same behavior for the request received during a CCH interval. We receive the request at about 13 ms, then wait for the SCH interval at 50 ms to switch to channel 172. After this, we wait 1 second, and return to alternating access at 50 ms past the next second.

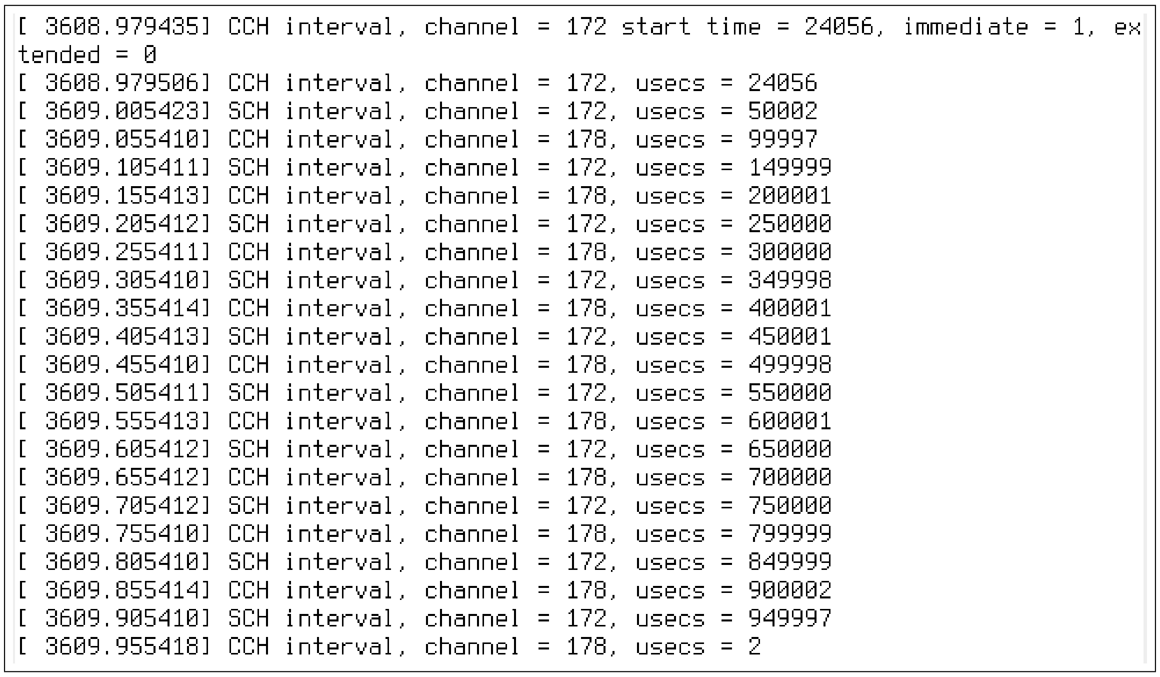
Figures 4.14 and 4.15 show the test output for immediate channel access. In both cases, we switch channels immediately, and return to alternating channel access at the next channel interval. In Figure 4.14, we receive the MLMEX-SCHSTART request at 92 ms, and we can see here that we switch to channel 172 at the same time. The next channel interval is a CCH interval at 100 ms. We return to alternating access here.

In Figure 4.15, we receive the request during a CCH interval, at about 24 ms, and we switch to channel 172 here are as well. Our next channel interval is the SCH interval at 50 ms. We return to alternating access here, but we stay on the service channel until the next CCH interval at 100 ms.



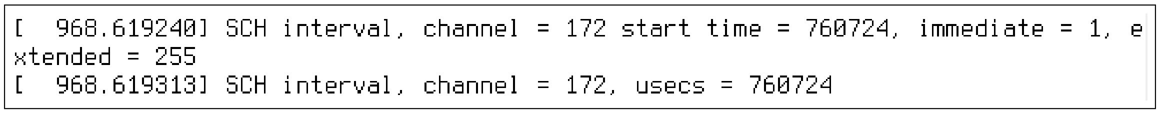
```
[ 3693.047652] SCH interval, channel = 172 start time = 92271, immediate = 1, ex
tended = 0
[ 3693.047752] SCH interval, channel = 172, usecs = 92271
[ 3693.055422] CCH interval, channel = 178, usecs = 100001
[ 3693.105412] SCH interval, channel = 172, usecs = 149999
[ 3693.155413] CCH interval, channel = 178, usecs = 200001
[ 3693.205411] SCH interval, channel = 172, usecs = 250000
[ 3693.255412] CCH interval, channel = 178, usecs = 300000
[ 3693.305411] SCH interval, channel = 172, usecs = 349999
[ 3693.355412] CCH interval, channel = 178, usecs = 400001
[ 3693.405411] SCH interval, channel = 172, usecs = 450000
[ 3693.455411] CCH interval, channel = 178, usecs = 499999
[ 3693.505410] SCH interval, channel = 172, usecs = 549999
[ 3693.555413] CCH interval, channel = 178, usecs = 600001
[ 3693.605410] SCH interval, channel = 172, usecs = 649998
[ 3693.655412] CCH interval, channel = 178, usecs = 700001
[ 3693.705411] SCH interval, channel = 172, usecs = 749999
[ 3693.755414] CCH interval, channel = 178, usecs = 800001
[ 3693.805414] SCH interval, channel = 172, usecs = 850000
[ 3693.855412] CCH interval, channel = 178, usecs = 900000
[ 3693.905409] SCH interval, channel = 172, usecs = 949998
[ 3693.955417] CCH interval, channel = 178, usecs = 1
[ 3694.005413] SCH interval, channel = 172, usecs = 50000
```

**Figure 4.14**: Immediate Access: MLMEX-SCHSTART received during SCH interval



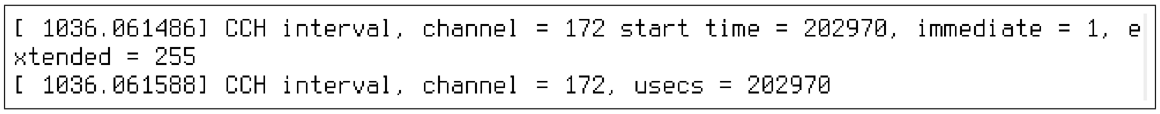
```
[ 3608.979435] CCH interval, channel = 172 start time = 24056, immediate = 1, ex
tended = 0
[ 3608.979506] CCH interval, channel = 172, usecs = 24056
[ 3609.005423] SCH interval, channel = 172, usecs = 50002
[ 3609.055410] CCH interval, channel = 178, usecs = 99997
[ 3609.105411] SCH interval, channel = 172, usecs = 149999
[ 3609.155413] CCH interval, channel = 178, usecs = 200001
[ 3609.205412] SCH interval, channel = 172, usecs = 250000
[ 3609.255411] CCH interval, channel = 178, usecs = 300000
[ 3609.305410] SCH interval, channel = 172, usecs = 349998
[ 3609.355414] CCH interval, channel = 178, usecs = 400001
[ 3609.405413] SCH interval, channel = 172, usecs = 450001
[ 3609.455410] CCH interval, channel = 178, usecs = 499998
[ 3609.505411] SCH interval, channel = 172, usecs = 550000
[ 3609.555413] CCH interval, channel = 178, usecs = 600001
[ 3609.605412] SCH interval, channel = 172, usecs = 650000
[ 3609.655412] CCH interval, channel = 178, usecs = 700000
[ 3609.705412] SCH interval, channel = 172, usecs = 750000
[ 3609.755410] CCH interval, channel = 178, usecs = 799999
[ 3609.805410] SCH interval, channel = 172, usecs = 849999
[ 3609.855414] CCH interval, channel = 178, usecs = 900002
[ 3609.905410] SCH interval, channel = 172, usecs = 949997
[ 3609.955418] CCH interval, channel = 178, usecs = 2
```

**Figure 4.15**: Immediate Access: MLMEX-SCHSTART received during CCH interval



```
[  968.619240] SCH interval, channel = 172 start time = 760724, immediate = 1, e
xtended = 255
[  968.619313] SCH interval, channel = 172, usecs = 760724
```

**Figure 4.16**: Immediate and continuous access: MLMEX-SCHSTART received during SCH interval



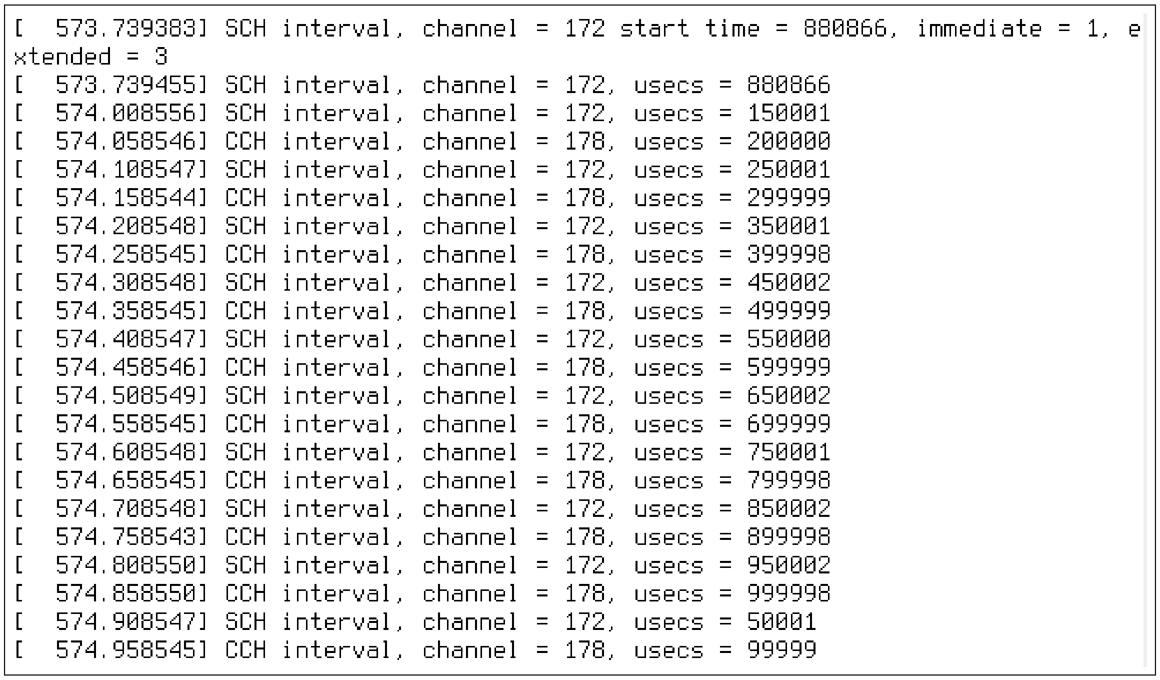
```
[ 1036.061486] CCH interval, channel = 172 start time = 202970, immediate = 1, e
xtended = 255
[ 1036.061588] CCH interval, channel = 172, usecs = 202970
```

**Figure 4.17**: Immediate and continuous access: MLMEX-SCHSTART received during CCH interval

Figures 4.16 and 4.17 show the case when we have both immediate and continuous access. Regardless of the interval that the MLMEX-SCHSTART request is received, the channel is switched immediately, at 760 ms in Figure 4.16, and at 202 ms in Figure 4.17. This is the only channel switch that happens here, since we are now in continuous mode.

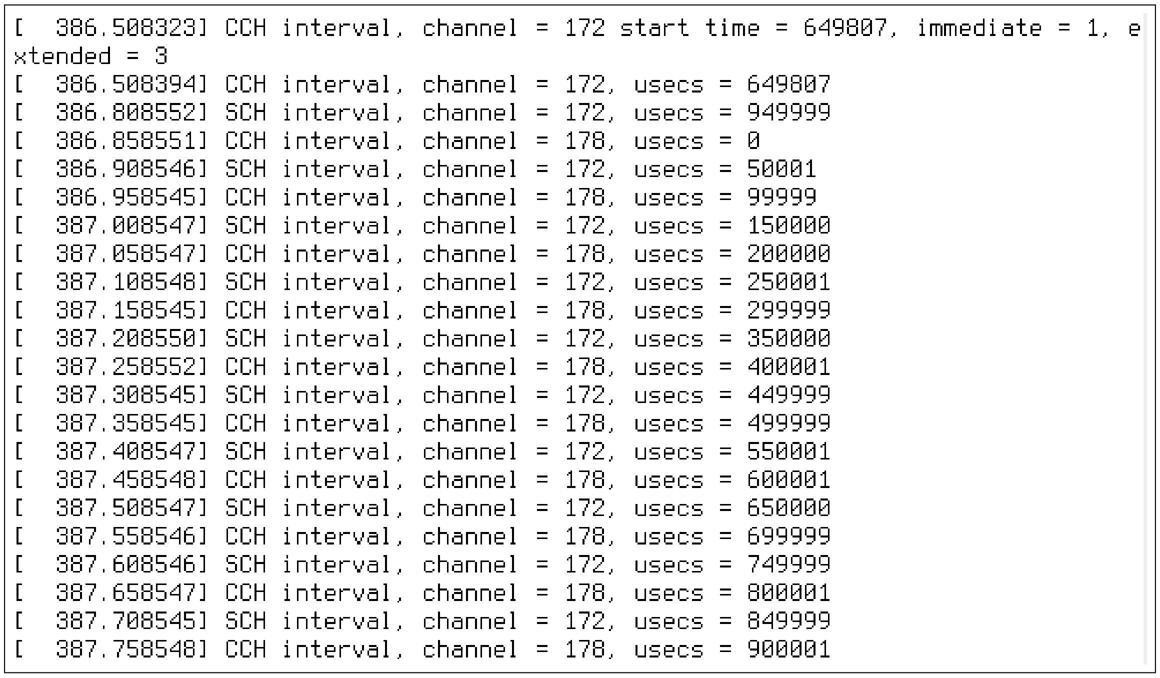
The Figures 4.18 and 4.19 show the cases where we have both immediate and extended access. These are identical to the cases shown in Figures 4.10 and 4.11 except that the first channel switch is done immediately, rather than at the following SCH interval. In Figure 4.18, the MLMEX-SCHSTART request is received at about 880 ms, during an SCH interval. The channel is switched to 172 immediately, then we wait for 3 CCH intervals to pass, 900 ms, 0 ms, and 100 ms. At 150 ms, we return to alternating access, switching back to the control channel at the following CCH interval, 200 ms.

In Figure 4.19, the MLMEX-SCHSTART request is received at the end of a CCH interval, at 649.8 ms. We switch channels immediately, then wait for the next 3 CCH intervals to pass, 700 ms, 800 ms, and 900 ms. At the following SCH interval, 950 ms, we return to alternating access. Our switch to the control channel is at 0 ms.



```
[  573.739383] SCH interval, channel = 172 start time = 880866, immediate = 1, e
xtended = 3
[  573.739455] SCH interval, channel = 172, usecs = 880866
[  574.008556] SCH interval, channel = 172, usecs = 150001
[  574.058546] CCH interval, channel = 178, usecs = 200000
[  574.108547] SCH interval, channel = 172, usecs = 250001
[  574.158544] CCH interval, channel = 178, usecs = 299999
[  574.208548] SCH interval, channel = 172, usecs = 350001
[  574.258545] CCH interval, channel = 178, usecs = 399998
[  574.308548] SCH interval, channel = 172, usecs = 450002
[  574.358545] CCH interval, channel = 178, usecs = 499999
[  574.408547] SCH interval, channel = 172, usecs = 550000
[  574.458546] CCH interval, channel = 178, usecs = 599999
[  574.508549] SCH interval, channel = 172, usecs = 650002
[  574.558545] CCH interval, channel = 178, usecs = 699999
[  574.608548] SCH interval, channel = 172, usecs = 750001
[  574.658545] CCH interval, channel = 178, usecs = 799998
[  574.708548] SCH interval, channel = 172, usecs = 850002
[  574.758543] CCH interval, channel = 178, usecs = 899998
[  574.808550] SCH interval, channel = 172, usecs = 950002
[  574.858550] CCH interval, channel = 178, usecs = 999998
[  574.908547] SCH interval, channel = 172, usecs = 50001
[  574.958545] CCH interval, channel = 178, usecs = 99999
```

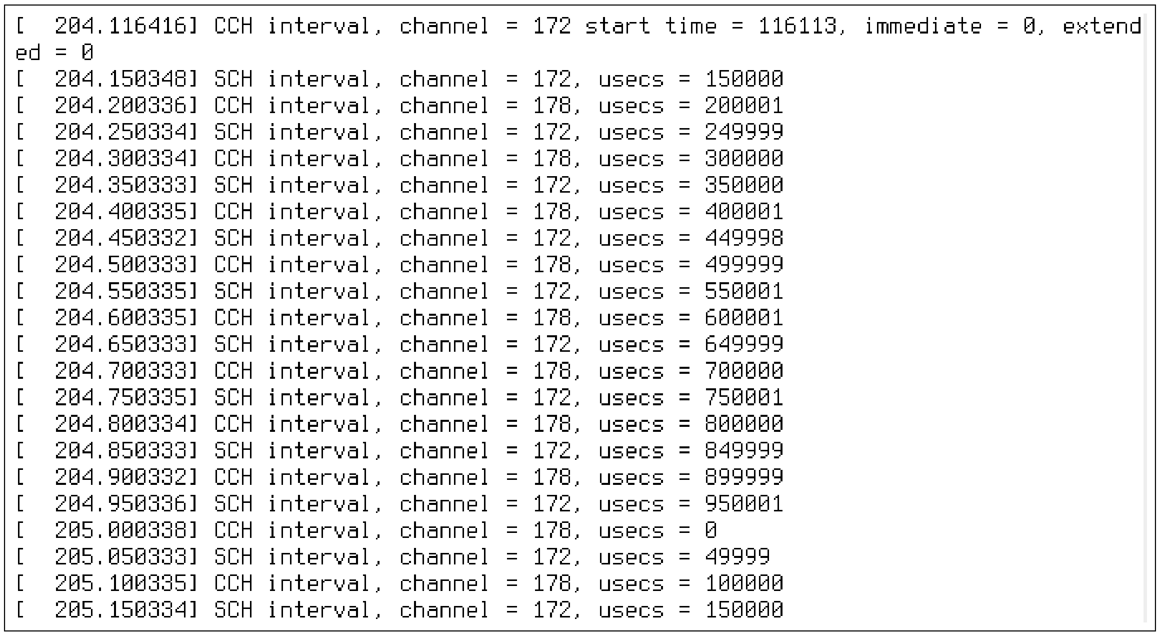
**Figure 4.18**: Immediate access and extended access = 3: MLMEX-SCHSTART received during SCH interval



```
[  386.508323] CCH interval, channel = 172 start time = 649807, immediate = 1, e
xtended = 3
[  386.508394] CCH interval, channel = 172, usecs = 649807
[  386.808552] SCH interval, channel = 172, usecs = 949999
[  386.858551] CCH interval, channel = 178, usecs = 0
[  386.908546] SCH interval, channel = 172, usecs = 50001
[  386.958545] CCH interval, channel = 178, usecs = 99999
[  387.008547] SCH interval, channel = 172, usecs = 150000
[  387.058547] CCH interval, channel = 178, usecs = 200000
[  387.108548] SCH interval, channel = 172, usecs = 250001
[  387.158545] CCH interval, channel = 178, usecs = 299999
[  387.208550] SCH interval, channel = 172, usecs = 350000
[  387.258552] CCH interval, channel = 178, usecs = 400001
[  387.308545] SCH interval, channel = 172, usecs = 449999
[  387.358545] CCH interval, channel = 178, usecs = 499999
[  387.408547] SCH interval, channel = 172, usecs = 550001
[  387.458548] CCH interval, channel = 178, usecs = 600001
[  387.508547] SCH interval, channel = 172, usecs = 650000
[  387.558546] CCH interval, channel = 178, usecs = 699999
[  387.608546] SCH interval, channel = 172, usecs = 749999
[  387.658547] CCH interval, channel = 178, usecs = 800001
[  387.708545] SCH interval, channel = 172, usecs = 849999
[  387.758548] CCH interval, channel = 178, usecs = 900001
```

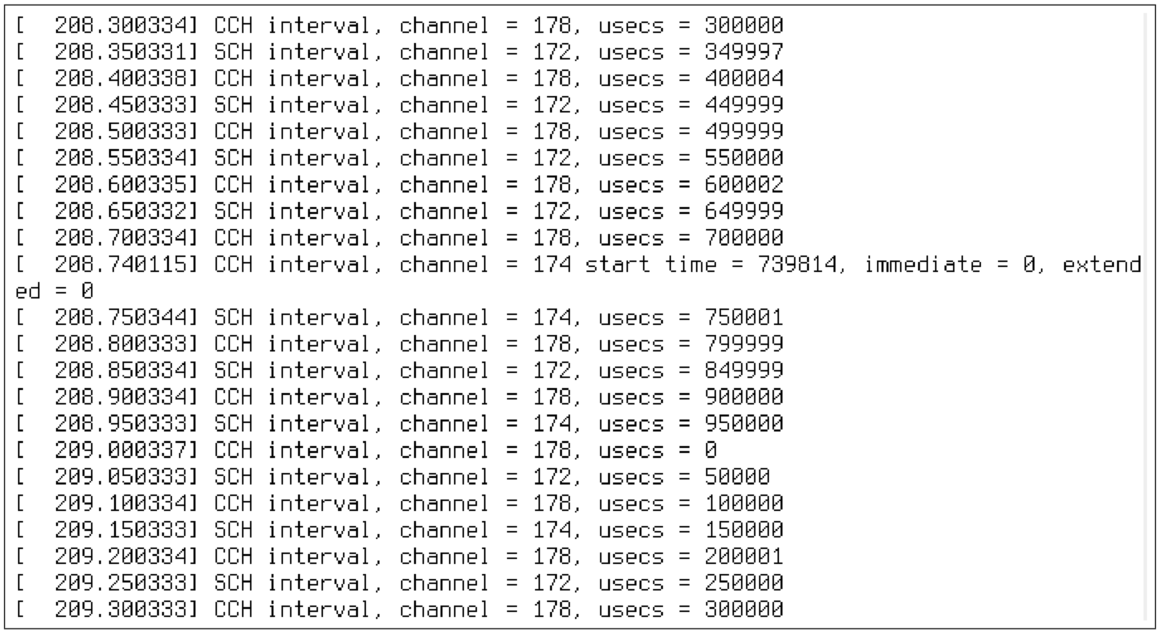
**Figure 4.19**: Immediate access and extended access = 3: MLMEX-SCHSTART received during CCH interval

Figures 4.20 through 4.25 show operation of multiple concurrent service channel access. Starting with figure 4.20, alternating service channel access is started between the control channel (178) and service channel 172. In Figure 4.21, alternating access to service channel 174 is added. After this MLMEX-SCHSTART request, we can see that we are switching to the control channel for every CCH interval, and alternating between service channels 172 and 174 during the SCH intervals. In Figure 4.22, access to service channel 176 is added in the same way. In Figures 4.25 through 4.23, access to service channels is ended in the reverse order that it has been added. In the last Figure, 4.23, when the MLMEX-SCHEND request is received, we return to continuous access on the control channel.



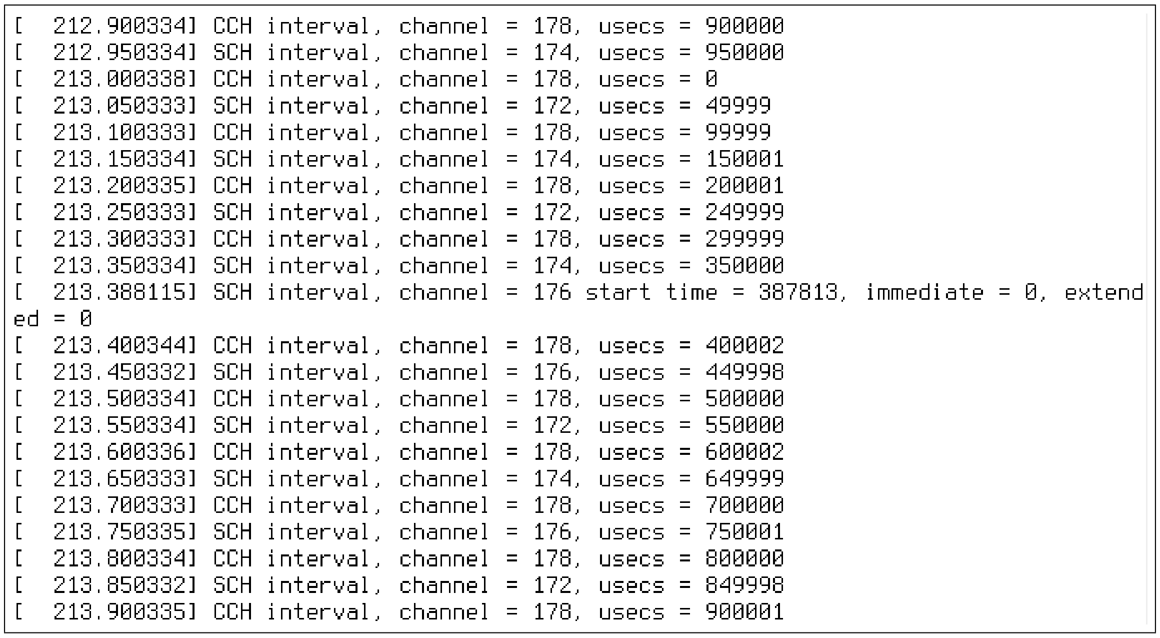
```
[  204.116416] CCH interval, channel = 172 start time = 116113, immediate = 0, extend
ed = 0
[  204.150348] SCH interval, channel = 172, usecs = 150000
[  204.200336] CCH interval, channel = 178, usecs = 200001
[  204.250334] SCH interval, channel = 172, usecs = 249999
[  204.300334] CCH interval, channel = 178, usecs = 300000
[  204.350333] SCH interval, channel = 172, usecs = 350000
[  204.400335] CCH interval, channel = 178, usecs = 400001
[  204.450332] SCH interval, channel = 172, usecs = 449998
[  204.500333] CCH interval, channel = 178, usecs = 499999
[  204.550335] SCH interval, channel = 172, usecs = 550001
[  204.600335] CCH interval, channel = 178, usecs = 600001
[  204.650333] SCH interval, channel = 172, usecs = 649999
[  204.700333] CCH interval, channel = 178, usecs = 700000
[  204.750335] SCH interval, channel = 172, usecs = 750001
[  204.800334] CCH interval, channel = 178, usecs = 800000
[  204.850333] SCH interval, channel = 172, usecs = 849999
[  204.900332] CCH interval, channel = 178, usecs = 899999
[  204.950336] SCH interval, channel = 172, usecs = 950001
[  205.000338] CCH interval, channel = 178, usecs = 0
[  205.050333] SCH interval, channel = 172, usecs = 49999
[  205.100335] CCH interval, channel = 178, usecs = 100000
[  205.150334] SCH interval, channel = 172, usecs = 150000
```

**Figure 4.20**: Multiple concurrent SCH access: Starting alternating channel access between the control channel and service channel 172



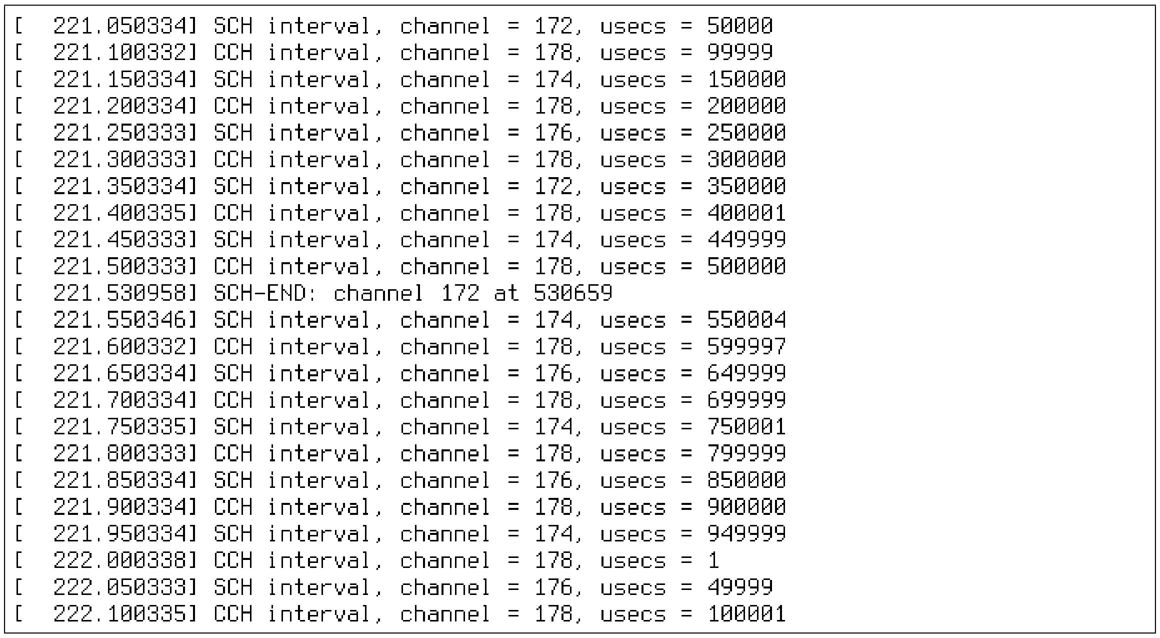
```
[  208.300334] CCH interval, channel = 178, usecs = 300000
[  208.350331] SCH interval, channel = 172, usecs = 349997
[  208.400338] CCH interval, channel = 178, usecs = 400004
[  208.450333] SCH interval, channel = 172, usecs = 449999
[  208.500333] CCH interval, channel = 178, usecs = 499999
[  208.550334] SCH interval, channel = 172, usecs = 550000
[  208.600335] CCH interval, channel = 178, usecs = 600002
[  208.650332] SCH interval, channel = 172, usecs = 649999
[  208.700334] CCH interval, channel = 178, usecs = 700000
[  208.740115] CCH interval, channel = 174 start time = 739814, immediate = 0, extend
ed = 0
[  208.750344] SCH interval, channel = 174, usecs = 750001
[  208.800333] CCH interval, channel = 178, usecs = 799999
[  208.850334] SCH interval, channel = 172, usecs = 849999
[  208.900334] CCH interval, channel = 178, usecs = 900000
[  208.950333] SCH interval, channel = 174, usecs = 950000
[  209.000337] CCH interval, channel = 178, usecs = 0
[  209.050333] SCH interval, channel = 172, usecs = 50000
[  209.100334] CCH interval, channel = 178, usecs = 100000
[  209.150333] SCH interval, channel = 174, usecs = 150000
[  209.200334] CCH interval, channel = 178, usecs = 200001
[  209.250333] SCH interval, channel = 172, usecs = 250000
[  209.300333] CCH interval, channel = 178, usecs = 300000
```

**Figure 4.21**: Multiple concurrent SCH access: Adding channel 174 to the existing alternating channel access between the control channel and service channel 172



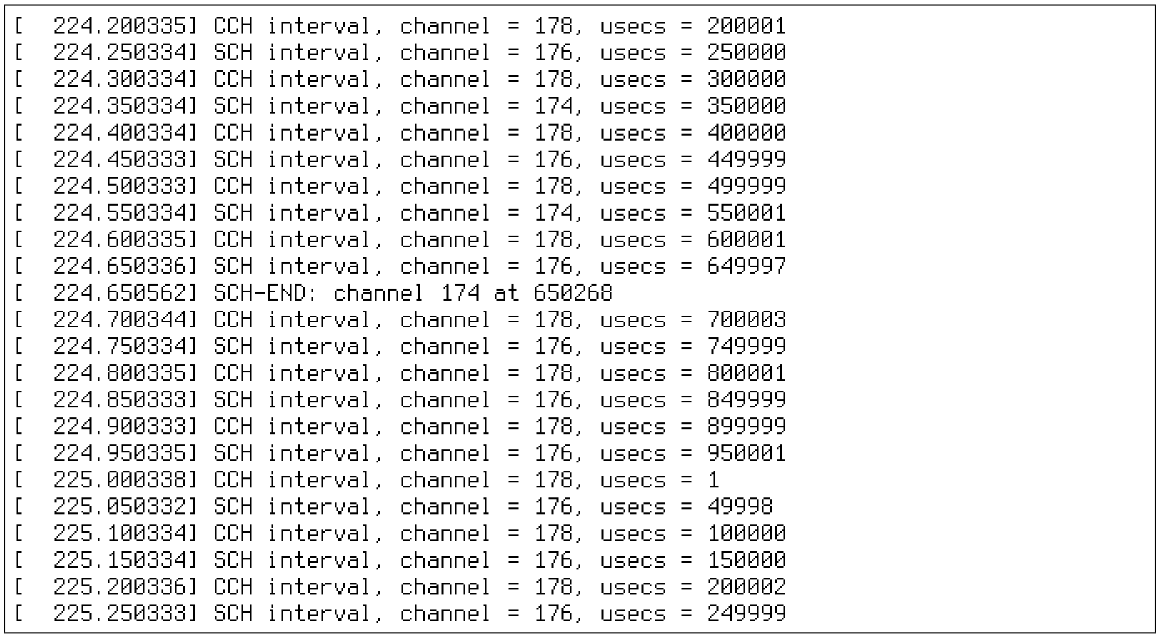
```
[  212.900334] CCH interval, channel = 178, usecs = 900000
[  212.950334] SCH interval, channel = 174, usecs = 950000
[  213.000338] CCH interval, channel = 178, usecs = 0
[  213.050333] SCH interval, channel = 172, usecs = 49999
[  213.100333] CCH interval, channel = 178, usecs = 99999
[  213.150334] SCH interval, channel = 174, usecs = 150001
[  213.200335] CCH interval, channel = 178, usecs = 200001
[  213.250333] SCH interval, channel = 172, usecs = 249999
[  213.300333] CCH interval, channel = 178, usecs = 299999
[  213.350334] SCH interval, channel = 174, usecs = 350000
[  213.388115] SCH interval, channel = 176 start time = 387813, immediate = 0, extend
ed = 0
[  213.400344] CCH interval, channel = 178, usecs = 400002
[  213.450332] SCH interval, channel = 176, usecs = 449998
[  213.500334] CCH interval, channel = 178, usecs = 500000
[  213.550334] SCH interval, channel = 172, usecs = 550000
[  213.600336] CCH interval, channel = 178, usecs = 600002
[  213.650333] SCH interval, channel = 174, usecs = 649999
[  213.700333] CCH interval, channel = 178, usecs = 700000
[  213.750335] SCH interval, channel = 176, usecs = 750001
[  213.800334] CCH interval, channel = 178, usecs = 800000
[  213.850332] SCH interval, channel = 172, usecs = 849998
[  213.900335] CCH interval, channel = 178, usecs = 900001
```

**Figure 4.22**: Multiple concurrent SCH access: Adding channel 176 to the existing alternating channel access between the control channel and service channels 172 and 174



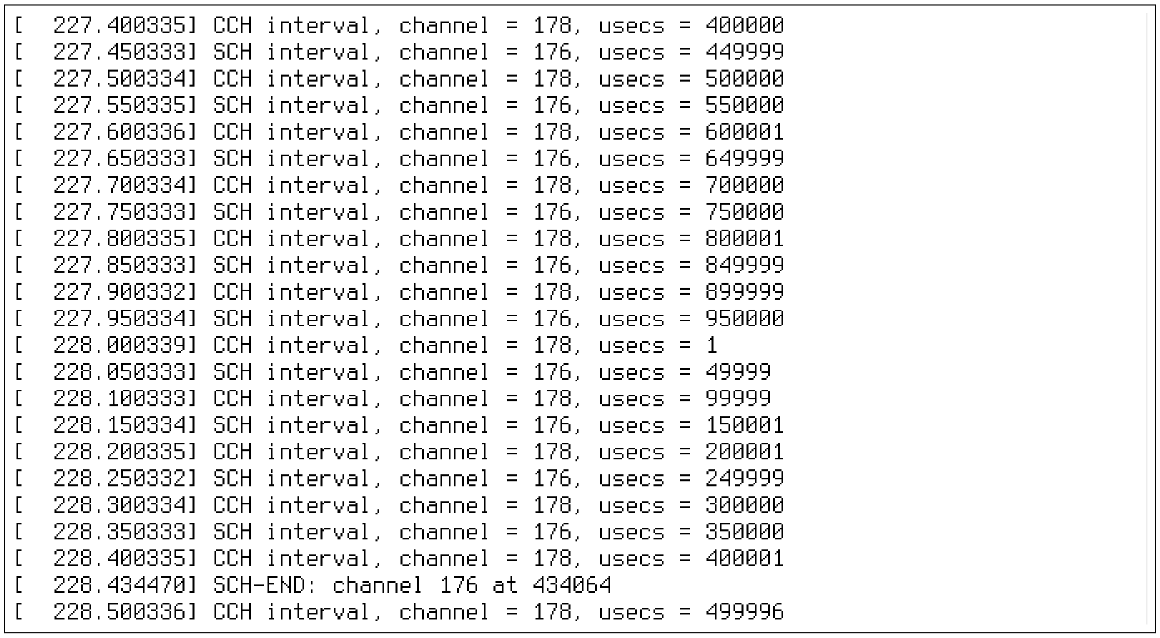
```
[  221.050334] SCH interval, channel = 172, usecs = 50000
[  221.100332] CCH interval, channel = 178, usecs = 99999
[  221.150334] SCH interval, channel = 174, usecs = 150000
[  221.200334] CCH interval, channel = 178, usecs = 200000
[  221.250333] SCH interval, channel = 176, usecs = 250000
[  221.300333] CCH interval, channel = 178, usecs = 300000
[  221.350334] SCH interval, channel = 172, usecs = 350000
[  221.400335] CCH interval, channel = 178, usecs = 400001
[  221.450333] SCH interval, channel = 174, usecs = 449999
[  221.500333] CCH interval, channel = 178, usecs = 500000
[  221.530958] SCH-END: channel 172 at 530659
[  221.550346] SCH interval, channel = 174, usecs = 550004
[  221.600332] CCH interval, channel = 178, usecs = 599997
[  221.650334] SCH interval, channel = 176, usecs = 649999
[  221.700334] CCH interval, channel = 178, usecs = 699999
[  221.750335] SCH interval, channel = 174, usecs = 750001
[  221.800333] CCH interval, channel = 178, usecs = 799999
[  221.850334] SCH interval, channel = 176, usecs = 850000
[  221.900334] CCH interval, channel = 178, usecs = 900000
[  221.950334] SCH interval, channel = 174, usecs = 949999
[  222.000338] CCH interval, channel = 178, usecs = 1
[  222.050333] SCH interval, channel = 176, usecs = 49999
[  222.100335] CCH interval, channel = 178, usecs = 100001
```

**Figure 4.23**: Multiple concurrent SCH access: Ending access to service channel 172



```
[   224.200335] CCH interval, channel = 178, usecs = 200001
[   224.250334] SCH interval, channel = 176, usecs = 250000
[   224.300334] CCH interval, channel = 178, usecs = 300000
[   224.350334] SCH interval, channel = 174, usecs = 350000
[   224.400334] CCH interval, channel = 178, usecs = 400000
[   224.450333] SCH interval, channel = 176, usecs = 449999
[   224.500333] CCH interval, channel = 178, usecs = 499999
[   224.550334] SCH interval, channel = 174, usecs = 550001
[   224.600335] CCH interval, channel = 178, usecs = 600001
[   224.650336] SCH interval, channel = 176, usecs = 649997
[   224.650562] SCH-END: channel 174 at 650268
[   224.700344] CCH interval, channel = 178, usecs = 700003
[   224.750334] SCH interval, channel = 176, usecs = 749999
[   224.800335] CCH interval, channel = 178, usecs = 800001
[   224.850333] SCH interval, channel = 176, usecs = 849999
[   224.900333] CCH interval, channel = 178, usecs = 899999
[   224.950335] SCH interval, channel = 176, usecs = 950001
[   225.000338] CCH interval, channel = 178, usecs = 1
[   225.050332] SCH interval, channel = 176, usecs = 49998
[   225.100334] CCH interval, channel = 178, usecs = 100000
[   225.150334] SCH interval, channel = 176, usecs = 150000
[   225.200336] CCH interval, channel = 178, usecs = 200002
[   225.250333] SCH interval, channel = 176, usecs = 249999
```

**Figure 4.24**: Multiple concurrent SCH access: Ending access to service channel 174



```
[   227.400335] CCH interval, channel = 178, usecs = 400000
[   227.450333] SCH interval, channel = 176, usecs = 449999
[   227.500334] CCH interval, channel = 178, usecs = 500000
[   227.550335] SCH interval, channel = 176, usecs = 550000
[   227.600336] CCH interval, channel = 178, usecs = 600001
[   227.650333] SCH interval, channel = 176, usecs = 649999
[   227.700334] CCH interval, channel = 178, usecs = 700000
[   227.750333] SCH interval, channel = 176, usecs = 750000
[   227.800335] CCH interval, channel = 178, usecs = 800001
[   227.850333] SCH interval, channel = 176, usecs = 849999
[   227.900332] CCH interval, channel = 178, usecs = 899999
[   227.950334] SCH interval, channel = 176, usecs = 950000
[   228.000339] CCH interval, channel = 178, usecs = 1
[   228.050333] SCH interval, channel = 176, usecs = 49999
[   228.100333] CCH interval, channel = 178, usecs = 99999
[   228.150334] SCH interval, channel = 176, usecs = 150001
[   228.200335] CCH interval, channel = 178, usecs = 200001
[   228.250332] SCH interval, channel = 176, usecs = 249999
[   228.300334] CCH interval, channel = 178, usecs = 300000
[   228.350333] SCH interval, channel = 176, usecs = 350000
[   228.400335] CCH interval, channel = 178, usecs = 400001
[   228.434470] SCH-END: channel 176 at 434064
[   228.500336] CCH interval, channel = 178, usecs = 499996
```

**Figure 4.25**: Multiple concurrent SCH access: Ending access to service channel 176

## 4.4 TIMING ADVERTISEMENT MANAGER

### 4.4.1 Debugfs

Testing of the Timing Advertisement Manager was done in a similar way to the Channel Scheduler, using debugfs entries to start and stop the transmission of timing advertisements. The following debugfs entries were created in the directory *timing_advertisements*:

**ta_start** This file is written to in order to initiate the start of timing advertisements. This corresponds to the MLMEX-TA primitive.

**ta_end** This file is written to in order to stop the transmission of timing advertisements. This corresponds to the MLMEX-TAEND primitive.

**log** This file can be written to in order to start and stop logging of received timing advertisements to the kernel log. This is used to verify that the correct number of timing advertisements are received. This can also be read from in order to see if we are currently logging received TAs.

The logging functionality for the Timing Advertisement Manager Debug entity logs the number of timing advertisements received every 5 seconds. This is used to verify that timing advertisements are received at the correct repeat rate specified in the MLMEX-TA request, since the repeat rate from the request indicates the number of TAs to be transmitted per 5 seconds.

### 4.4.2 Testing Procedure

When a timing advertisement is received, the system time is updated using the *mib_set_utc_time()* function. The testing of this functionality was explained in Section 4.2 of this chapter. We also need to verify that the timing advertisemnts were transmitted and received at the correct repeat rate. This was done using two of our devices, one for transmission and one for reception. The testing procedure for this is as follows:

1. On the receiving device, "start" is written to the *log* debugfs entry. This starts a periodic timer with an expiry period of 5 seconds. Received TAs are counted, and every time the timer expires, the number of received TAs is printed to the kernel log. The count is reset each time the timer expires.

2. On the sending device, the *ta_start* debugfs entry is written to in order to start transmission of timing advertisements.

3. On the receiving device, the kernel logs are checked in order to verify that the number of received timing advertisements matches the repeat rate that was specified in the previous step.

This procedure was repeated for various repeat rates, for both continuous channel access, and alternating channel access. For alternating channel access, we verified that the timing advertisements were received on the same channel that it was specified that they were sent on. This was done by:

1. Start logging on the receiving device.

2. Initiate alternating channel access between a service channel and a control channel on the transmitting device, then start transmission of timing advertisements on the service channel.

3. On the receiving device, set channel access to continuous on the control channel. Verify that we are not receiving any of the transmitted TAs.

4. On the receiving device, set channel access to continuous on the service channel on which the TAs are being transmitted. Verify that we are receiving the correct number of TAs per 5 seconds.

5. On the receiving device, set channel access to alternating on both the control channel and the service channel on which the TAs are being transmitted. Verify that we are receiving the same number of TAs as in the previous step, and that this matches the number specified in the repeat rate.

After performing this procedure, it was determined that timing advertisements were being transmitted and received on the channel specified in the MLMEX-TA request.

### 4.4.3 Results

The number of received timing advertisements per 5 seconds closely matched the sepcified repeat rate while transmitting during both continuous and alternating channel access modes. The repeat rates tested were: 0 (sends only a single TA), 1, 50, 100, 100, and 255. In both cases, continuous and alternating access, the results were identical. The number of received timing advertisements for all cases matched the desired repeat rate exactly.

The output from these tests is shown in Figures 4.26 - 4.29 for continuous channel access, and in Figures 4.30 - 4.33 for alternating channel access. Also, note that in many cases, logging of received timing advertisements was started prior to transmission. Because of this, the output shows a lower number of received TAs at the beginning, but the numbers are consistent after this interval.

```
[ 3194.971218] Starting TA debug timer
[ 3200.971237] Number of TAs received in past 5s: 1
[ 3205.971240] Number of TAs received in past 5s: 1
[ 3210.971239] Number of TAs received in past 5s: 1
[ 3215.971239] Number of TAs received in past 5s: 1
[ 3220.971239] Number of TAs received in past 5s: 1
[ 3225.971238] Number of TAs received in past 5s: 1
[ 3230.971239] Number of TAs received in past 5s: 1
```

**Figure 4.26**: Received TAs per 5 s, continuous access: repeat rate = 1

```
[  650.495190] Starting TA debug timer
[  656.495211] Number of TAs received in past 5s: 0
[  661.495209] Number of TAs received in past 5s: 40
[  666.495209] Number of TAs received in past 5s: 50
[  671.495210] Number of TAs received in past 5s: 50
[  676.495210] Number of TAs received in past 5s: 50
[  681.495209] Number of TAs received in past 5s: 50
[  686.495209] Number of TAs received in past 5s: 50
[  691.495209] Number of TAs received in past 5s: 50
```

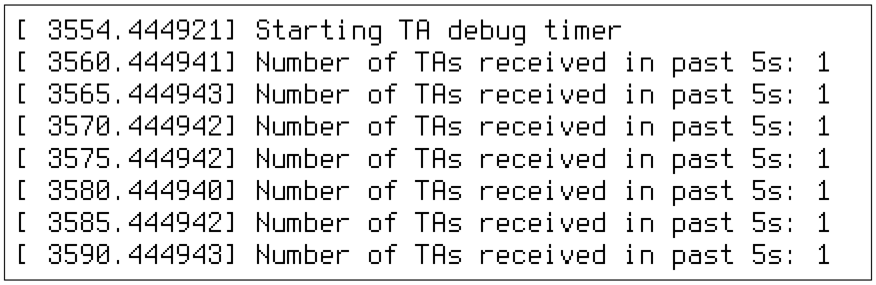
**Figure 4.27**: Received TAs per 5 s, continuous access: repeat rate = 50

```
[  743.463281] Starting TA debug timer
[  749.463301] Number of TAs received in past 5s: 0
[  754.463302] Number of TAs received in past 5s: 80
[  759.463301] Number of TAs received in past 5s: 101
[  764.463300] Number of TAs received in past 5s: 100
[  769.463299] Number of TAs received in past 5s: 100
[  774.463300] Number of TAs received in past 5s: 100
[  779.463299] Number of TAs received in past 5s: 100
[  784.463300] Number of TAs received in past 5s: 100
```

**Figure 4.28**: Received TAs per 5 s, continuous access: repeat rate = 100

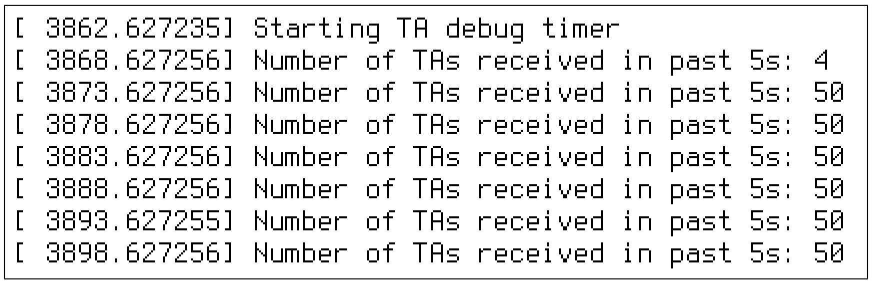
```
[  154.208548] Starting TA debug timer
[  159.208570] Number of TAs received in past 5s: 0
[  164.208574] Number of TAs received in past 5s: 147
[  169.208566] Number of TAs received in past 5s: 255
[  174.208568] Number of TAs received in past 5s: 255
[  179.208568] Number of TAs received in past 5s: 255
[  184.208568] Number of TAs received in past 5s: 255
[  189.208567] Number of TAs received in past 5s: 255
[  194.208568] Number of TAs received in past 5s: 255
```

**Figure 4.29**: Received TAs per 5 s, continuous access: repeat rate = 255



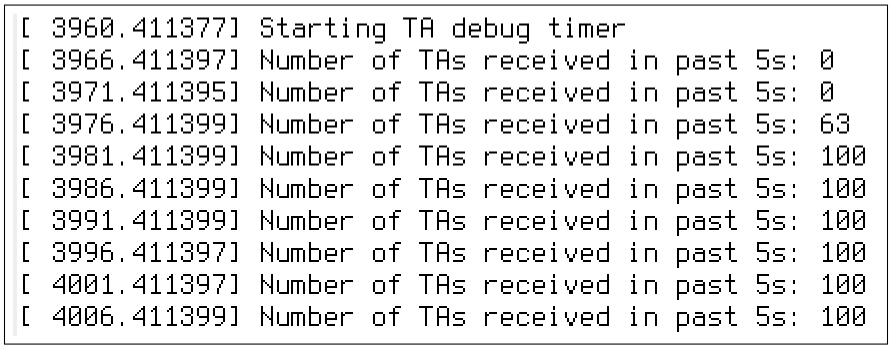
```
[ 3554.444921] Starting TA debug timer
[ 3560.444941] Number of TAs received in past 5s: 1
[ 3565.444943] Number of TAs received in past 5s: 1
[ 3570.444942] Number of TAs received in past 5s: 1
[ 3575.444942] Number of TAs received in past 5s: 1
[ 3580.444940] Number of TAs received in past 5s: 1
[ 3585.444942] Number of TAs received in past 5s: 1
[ 3590.444943] Number of TAs received in past 5s: 1
```

**Figure 4.30**: Received TAs per 5 s, alternating access: repeat rate = 1



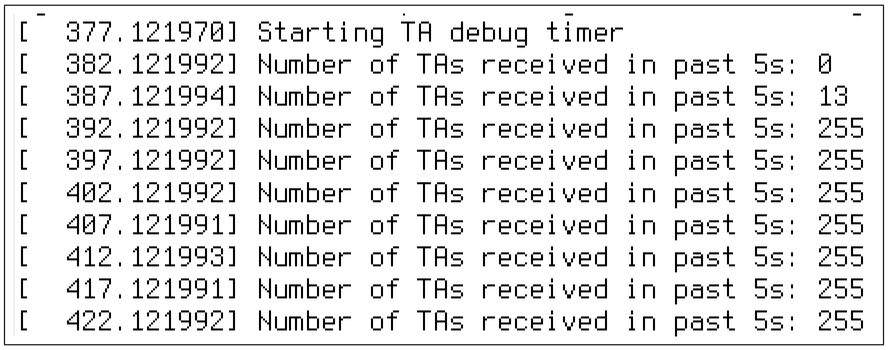
```
[ 3862.627235] Starting TA debug timer
[ 3868.627256] Number of TAs received in past 5s: 4
[ 3873.627256] Number of TAs received in past 5s: 50
[ 3878.627256] Number of TAs received in past 5s: 50
[ 3883.627256] Number of TAs received in past 5s: 50
[ 3888.627256] Number of TAs received in past 5s: 50
[ 3893.627255] Number of TAs received in past 5s: 50
[ 3898.627256] Number of TAs received in past 5s: 50
```

**Figure 4.31**: Received TAs per 5 s, alternating access: repeat rate = 50



```
[ 3960.411377] Starting TA debug timer
[ 3966.411397] Number of TAs received in past 5s: 0
[ 3971.411395] Number of TAs received in past 5s: 0
[ 3976.411399] Number of TAs received in past 5s: 63
[ 3981.411399] Number of TAs received in past 5s: 100
[ 3986.411399] Number of TAs received in past 5s: 100
[ 3991.411399] Number of TAs received in past 5s: 100
[ 3996.411397] Number of TAs received in past 5s: 100
[ 4001.411397] Number of TAs received in past 5s: 100
[ 4006.411399] Number of TAs received in past 5s: 100
```

**Figure 4.32**: Received TAs per 5 s, alternating access: repeat rate = 100



```
[  377.121970] Starting TA debug timer
[  382.121992] Number of TAs received in past 5s: 0
[  387.121994] Number of TAs received in past 5s: 13
[  392.121992] Number of TAs received in past 5s: 255
[  397.121992] Number of TAs received in past 5s: 255
[  402.121992] Number of TAs received in past 5s: 255
[  407.121991] Number of TAs received in past 5s: 255
[  412.121993] Number of TAs received in past 5s: 255
[  417.121991] Number of TAs received in past 5s: 255
[  422.121992] Number of TAs received in past 5s: 255
```

**Figure 4.33**: Received TAs per 5 s, alternating access: repeat rate = 255

## 4.5 TRANSMIT OPERATION

In order to test the overall transmit and receive operation, a comprehensive integration test was performed. This test was performed using two of our devices, one including the daughterboard with GPS, and one with only the main board. This test included the use of the UTC synchronization functionality, timing advertisements, the transmitter profile table, and data transmission for both WSMP and IPv6 data paths. Since the layers of the WAVE stack above 1609.4 are not implemented yet, management functions were invoked by writing to the debugfs entries. For data transmission, raw link layer sockets were used for testing both WSMP and IPv6 paths.

### 4.5.1 Challenges

*Transmitting from the Application Layer*

As mentioned previously, the layers above 1609.4 in the WAVE protocol stack have not been implemented yet. However, the transmit functionality still had to be tested for both the WSMP and the IPv6 data paths. In order to do this, our test application used raw link-layer sockets to create data packets to be passed down to the 1609.4 module. When the socket is created, the protocol is specified as either WSMP or IPv6 by passing the corresponding value to the *protocol* parameter. This value will be located in the *ethertype* field of the packets sent on this socket.

A struct containing transmit information used in a WSMP header was defined. The most important piece of information here is the channel number. When the packets to be used on the control channel are created, the channel number field is assigned the index of the control channel. When these packets reach the 1609.4 module, this header is parsed and the data is inserted to the underlying *sk_buff*'s control buffer.

For the IPv6 data path, this same information was included in the packet, but only for debugging purposes. When IPv6 packets reach the 1609.4 module, the transmit parameters are retrieved from a transmitter profile, stored in the MIB. In order to ensure that the data identified as IPv6 is getting its transmit information from the MIB, the test was run without first registering a transmitter profile. The results were as expected: when there is no transmitter profile registered, the packets were not transmitted.

### *Time Synchronization*

The purpose of this test was to show that both IPv6 and WSMP data can be transmitted on the correct channels during alternating channel access. Earlier attempts involved first synchronizing the devices using timing advertisements, since at the moment, we only have one daughterboard with GPS capability. After the devices were synchronized, timing advertisements were stopped, then alternating channel access was started on both devices. Once both devices were switching channels, data packets were created at the application layer to be transmitted on the control channel and the service channel. As the same test was run multiple times, the packet delivery ratio dropped significantly. It turns out, that even though the devices were time synchronized at the beginning, the clocks would begin to drift apart. This is because the device with GPS is constantly receiving an updated time, while the other device is not.

In order to overcome this, it was decided that the receiving device would broadcast timing advertisements at a relatively low repeat rate, while it is listening for packets. This way, the transmitting device would constantly be updating its time in order to stay in sync with the receiving device.

### 4.5.2 Testing Procedure

In this scenario, the device with GPS capabilities is used as the receiver, and the device without GPS is the transmitter. The procedure is as follows, and is shown in Figure 4.34.

1. Both devices start out being relatively synchronized. Both devices begin channel access alternating between the control channel and service channel 180.

2. Device A (with GPS) begins broadcasting timing advertisements on the service channel at a relatively low frequency, as not to add too much overhead to the channel. (Previous tests were done without this step, and synchronization was lost due to clock drifting)

3. Throughout the test, device B receives the timing advertisements and updates its UTC time, using the *mib_set_utc_time()* function.

4. Device B registers a transmitter profile for channel 180 to be used for IPv6 data.

5. Device B begins transmitting"WSMP" data on the control channel, and "IPv6" data on the service channel. Since the upper layers are not implemented yet, the packet contents are arbitrary, containing information needed for testing. Among the contents of the packets, is a sequence number, used for calculating the delivery ratio.

6. Device A receives packets on the control channel and the service channel. As a packet is received, it is counted (per channel), and the count is compared with the sequence number. When the sequence number is greater than 1000, the application prints the delivery ratio to a file.

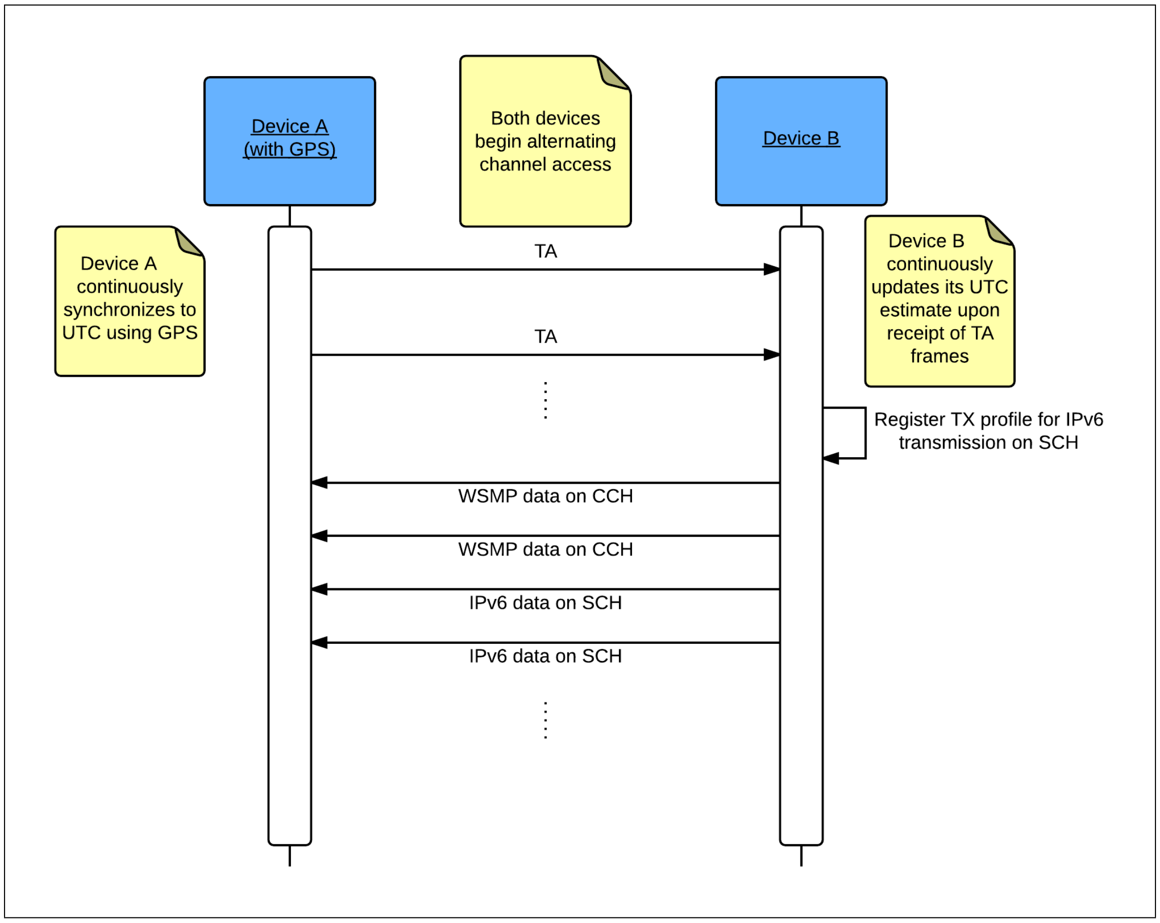

Device A (with GPS)
Both devices begin alternating channel access
Device B
Device A continuously synchronizes to UTC using GPS
Device B continuously updates its UTC estimate upon receipt of TA frames
TA
TA
Register TX profile for IPv6 transmission on SCH
WSMP data on CCH
WSMP data on CCH
IPv6 data on SCH
IPv6 data on SCH

**Figure 4.34**: Testing Procedure for Transmit Operation

### 4.5.3 Test Application

A userspace application was created in order to implement the above testing procedure. In this application, 2 sockets were created, one specifies WSMP as its protocol and is used for the control channel. The other specifies IPv6, and is used for the service channel. Then we start alternating channel access on the control channel and service channel 180 by writing to the debugfs entry *sch_start*. The program then prompts the user to specify whether it is running on the transmitting or receiving device.

If we are on the receiving device, the transmission of timing advertisements begins on the service channel. This is done by writing to the *ta_start* debugfs entry, since these are management frames, and not visible to the application layer. Once the TAs have begun, we create 2 threads, using the *pthread* library. One thread receives from the control channel socket, while the other receives from the service channel socket. In these receive threads, we count the number of packets as they are received. Once a packet with a sequence number of 1000 or higher has been received, the delivery ratio is calculated by dividing the number of received packets by the sequence number of the last received packet. This ratio is written to a file, then the thread exits. Once both receive threads exit, timing advertisements are stopped, service channel access is ended, and the debugfs files are closed.

If we are on the transmitting device, a transmitter profile is registered using the debugfs entry *reg_tx_profile*. The the user is prompted for packet intervals for both the service channel and the control channel. These packet intervals determine the frequency that the packets will be created and sent by the application. After this, 2 threads are created for transmitting packets, one for each channel. Each of the transmit threads waits the appropriate interval, then creates and transmits packets on the appropriate socket. Once 1500 packets have been transmitted on a thread, the thread exits.

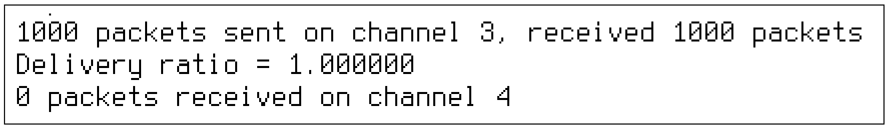
Prior to running the application for the first time, the devices were synchronized by transmitting a timing advertisement on the control channel. This ensures that once alternating channel access begins, the timing advertisements on the service channel are received. The devices then stay in sync while the application is running by using these timing advertisements.

### 4.5.4 Results

This test was run several times, sometimes with modifications, in order to ensure that the transmit and receive operation os the 1609.4 module work as intended. The items verified using this test are as follows:

1. Data transmitted on the control channel is received on the control channel.
2. Data transmitted on a service channel is received on the correct service channel
3. Data that contains an *ethertype* value of 0x88DC (WSMP) is transmitted on the channel specified in its header.
4. Data that contains an *ethertype* value of 0x88DD (IPv6) is transmitted on the channel retrieved from the MIB.

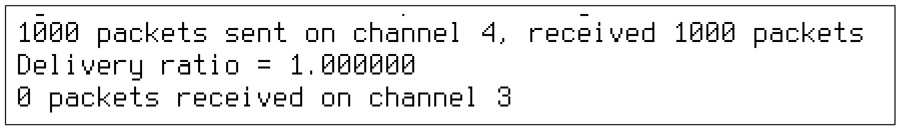
In order to verify that data is transmitted on the correct channel during alternating channel access, the test above was modified for the receiving device. In this case, the receiver stays on one channel while the transmitter is alternating between the control channel and a service channel. In the application, data is being transmitted on both channels. We should only receive data being transmitted on the receiver's channel. This was verified for both the control channel and a service channel, using both IPv6 data and WSMP data types. In order to show that we have not received any packets on the channel that the receiving device is not switching to, the receiving thread for that channel was modified to wait until the other channel's receiving thread receives 1000 packets, and then print out the number received, which should be 0.



```
1000 packets sent on channel 3, received 1000 packets
Delivery ratio = 1.000000
0 packets received on channel 4
```

**Figure 4.35**: Number of received packets for SCH and CCH while the receiving device is in continuous mode on the CCH. Channel 3 is the index of the CCH (178) and channel 4 is the index of the SCH used (180).

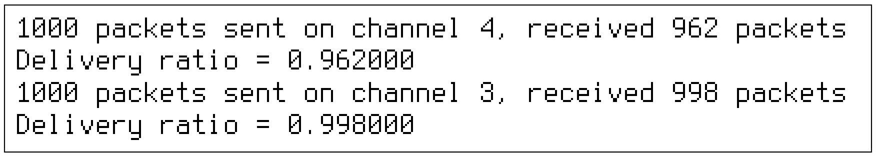
Figure 4.35 shows the output of this test while the receiver stays on the control channel in continuous access mode. This shows that the data transmitted on the control channel is received as expected, and that the data on the service channel is not received.



```
1000 packets sent on channel 4, received 1000 packets
Delivery ratio = 1.000000
0 packets received on channel 3
```

**Figure 4.36**: Number of received packets for SCH and CCH while the receiving device is in continuous mode on the SCH. Channel 3 is the index of the CCH (178) and channel 4 is the index of the SCH used (180).

Figure 4.36 shows the output of the same test, but in this case the receiving device is in continuous mode on the service channel. These two tests show that data being transmitted on the control channel is received on the control channel, and data being transmitted on the service channel is received on the service channel.



```
1000 packets sent on channel 4, received 962 packets
Delivery ratio = 0.962000
1000 packets sent on channel 3, received 998 packets
Delivery ratio = 0.998000
```

**Figure 4.37**: Number of received packets for SCH and CCH while the receiving device is alternating between the CCH and SCH. Channel 3 is the index of the CCH (178) and channel 4 is the index of the SCH used (180).

Figure 4.37 shows the output of the test where we are using alternating channel access on both the transmitting and receiving device. This shows that packets transmitted on both channels are received on both channels. Figure 4.38 shows the output of the same test as in Figure 4.37, but here we do not register a transmitter profile for the service channel prior to transmission. Because of this, the packets are not transmitted on the service channel. This shows that the IPv6 packets are sent on the channel specified in the transmitter profile, and they are not transmitted if a profile has not been registered.

```
1000 packets sent on channel 3, received 998 packets
Delivery ratio = 0.998000
0 packets received on channel 4
```

**Figure 4.38**: Number of received packets for SCH and CCH while the receiving device is alternating between the CCH and SCH, but a transmitter profile has not been registered for service channel 180 (index 4). Channel 3 is the index of the CCH (178) and channel 4 is the index of the SCH used (180).

# CHAPTER 5
# CONCLUSION

Vehicular communication has the potential to greatly impact society by offering solutions to driving-related problems of both safety and convenience. VANET has the capability to help alleviate traffic congestion, warn drivers of impending collisions, and can provide useful information and entertainment to passengers. We are at a critical time for research in this area, as the IEEE WAVE set of standards was recently released, and the United States Department of Transportation is working on regulations mandating the use of vehicular communication by 2017 [27]. The IEEE WAVE set of standards defines standardized technology that will enable the formation of these vehicular networks. The 1609.4 standard in particular provides the channel switching functionality that allows vehicles with a single radio to participate in both safety and non-safety services concurrently.

In this thesis, we presented an implementation of IEEE 1609.4 that will serve as a key component in our Smart Drive VANET testbed. This implementation includes 1609.4's channel switching and timing synchronization functionalities, as well as concurrent access to multiple service channels. The channel switching functionality implemented includes all of the specified channel access modes, with switching compliant to 50 ms interval boundaries. Timing synchronization includes updating of the system's UTC time estimate as well as sending timing information over the air using the Timing Advertisement mechanism. The addition of multiple concurrent service channel access allows a vehicle to participate in more than one service concurrently in addition to maintaining control channel access.

We have shown that this implementation has been tested thoroughly, and that all implemented features perform as intended. The work presented here serves as a solid foundation for our testbed, and proves invaluable to future research in the area of vehicular networking.

## 5.1 FUTURE WORK

As this implementation is just one element in the WAVE protocol stack, and in the larger Smart Drive VANET testbed, there are several areas for future reaseach to be conducted:

- Implementation of optional 1609.4 features, such as Vendor Specific Action frames.
- Implementation of the remaining layers of the WAVE protocol stack, including the IEEE 1609.3 and IEEE 1609.2 standards.
- Performance evaluation of the WAVE standards in on-road, real-world conditions, conducted with the Smart Drive testbed.
- Identification of possible improvements or revisions to the WAVE standards based on our implementations and evaluations.
- Implementation of various VANET routing protocols and applications, built on top of the WAVE protocol stack, and the evaluation of these protocols and applications using the Smart Drive testbed.

# BIBLIOGRAPHY

[1] Ameixieira, C.; Matos, J.; Moreira, R.; Cardote, A; Oliveira, A; Sargento, S., "An IEEE 802.11p/WAVE implementation with synchronous channel switching for seamless dual-channel access (poster)," Vehicular Networking Conference (VNC), 2011 IEEE , vol., no., pp.214,221, 14-16 Nov. 2011

[2] Ali J. Ghandour, Marco Di Felice, Luciano Bononi, and Hassan Artail. 2012. Modeling and simulation of WAVE 1609.4-based multi-channel vehicular ad hoc networks. In Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques (SIMUTOOLS '12). ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 148-156.

[3] Grafling, S.; Mahonen, P.; Riihijarvi, J., "Performance evaluation of IEEE 1609 WAVE and IEEE 802.11p for vehicular communications," Ubiquitous and Future Networks (ICUFN), 2010 Second International Conference on , vol., no., pp.344,348, 16-18 June 2010

[4] Biddlestone, S.; Redmill, K.A, "A GNU Radio based testbed implementation with IEEE 1609 WAVE functionality," Vehicular Networking Conference (VNC), 2009 IEEE , vol., no., pp.1,7, 28-30 Oct. 2009

[5] Kai-Yun Ho; Po-Chun Kang; Chung-Hsien Hsu; Ching-Hai Lin, "Implementation of WAVE/DSRC Devices for vehicular communications," Computer Communication Control and Automation (3CA), 2010 International Symposium on , vol.2, no., pp.522,525, 5-7 May 2010

[6] Di Felice, M.; Ghandour, AJ.; Artail, H.; Bononi, L., "Enhancing the performance of safety applications in IEEE 802.11p/WAVE Vehicular Networks," World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2012 IEEE International Symposium on a , vol., no., pp.1,9, 25-28 June 2012

[7] IEEE Guide for Wireless Access in Vehicular Environments (WAVE) - Architecture," IEEE Std 1609.0-2013 , vol., no., pp.1,78, March 5 2014

[8] IEEE Standard for Wireless Access in Vehicular Environments (WAVE)–Multi-channel Operation," IEEE Std 1609.4-2010 (Revision of IEEE Std 1609.4-2006) , vol., no., pp.1,89, Feb. 7 2011

[9] IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 11: Wireless LAN Medium Access Control

(MAC) and Physical Layer (PHY) Specifications Amendment 6: Wireless Access in Vehicular Environments," IEEE Std 802.11p-2010 (Amendment to IEEE Std 802.11-2007 as amended by IEEE Std 802.11k-2008, IEEE Std 802.11r-2008, IEEE Std 802.11y-2008, IEEE Std 802.11n-2009, and IEEE Std 802.11w-2009) , vol., no., pp.1,51, July 15 2010

[10] IEEE Standard for Wireless Access in Vehicular Environments (WAVE) - Networking Services," IEEE Std 1609.3-2010 (Revision of IEEE Std 1609.3-2007) , vol., no., pp.1,144, Dec. 30 2010

[11] Xiaoyi Zhang; Hongyu An; Zhen Qin; Bin Xiang; Lin Zhang, "Implementation and field test of a small-scale WAVE system in IEEE 802.11p V2R/V2V environments," Communication Technology (ICCT), 2013 15th IEEE International Conference on , vol., no., pp.462,466, 17-19 Nov. 2013

[12] Ghosh, S.; Kundu, A; Jana, D., "Implementation challenges of time synchronization in vehicular networks," Recent Advances in Intelligent Computational Systems (RAICS), 2011 IEEE , vol., no., pp.575,580, 22-24 Sept. 2011

[13] About Vim: vim online. [Online] http://www.vim.org/about.php

[14] Embedded Linux system development. [Online] http://free-electrons.com/doc/training/embedded-linux/embedded-linux-slides.pdf

[15] NFS-Root mini-HOWTO. [Online] http://www.tldp.org/HOWTO/NFS-Root.html

[16] Corbet J.; Rubini A.; Kroah-Hartman G., "Linux Device Drivers," O'Reilly Media Inc., 2005

[17] Benvenuti C., "Understanding Linux Network Internals," O'Reilly Media Inc., 2006

[18] mac80211 - Linux Wireless. [Online] http://wireless.kernel.org/en/developers/Documentation/mac80211

[19] Horstmann C., "Object-Oriented Design & Patterns," John Wiley & Sons, Inc., 2006

[20] Kernel APIs, Part 3: Timers and lists in the 2.6 kernel. [Online] http://www.ibm.com/developerworks/library/l-timers-list/

[21] Workqueues. [Online] http://www.makelinux.net/ldd3/chp-7-sect-6

[22] Top and Bottom Halves. [Online] http://www.makelinux.net/ldd3/chp-10-sect-4

[23] Eze, Elias C.; Zhang, Sijing; Liu, Enjie, ”Vehicular ad hoc networks (VANETs): Current state, challenges, potentials and way forward,” Automation and Computing (ICAC), 2014 20th International Conference on , vol., no., pp.176,181, 12-13 Sept. 2014

[24] Ree, S.; Alins, J.; Mata-Diaz, J.; Gaan, C.; Muoz, J.L.; Esparza, O., ”Vespa: Emulating Infotainment Applications in Vehicular Networks,” Pervasive Computing, IEEE , vol.13, no.3, pp.58,66, July-Sept. 2014

[25] Smart Mobile Computing - NSF MRI Grant. [Online] http://smc.eng.fau.edu/#/nsf-mri-grant/

[26] Cadger, F.; Curran, K.; Santos, J.; Moffett, S., ”A Survey of Geographical Routing in Wireless Ad-Hoc Networks,” Communications Surveys & Tutorials, IEEE , vol.15, no.2, pp.621,653, Second Quarter 2013

[27] U.S. may mandate ’talking’ cars by early 2017. [Online] http://www.reuters.com/article/2014/02/03/us-autos-technology-rules-idUSBREA1218M20140203

[28] President Obama Calls for Vehicle-to-Vehicle, Vehicle-to-Infrastructure Communication. [Online] http://www.govtech.com/federal/President-Obama-Calls-for-Vehicle-to-Vehicle-Vehicle-to-Infrastructure-Communication-GT.html

[29] Debugging the Linux Kernel with debugfs. [Online] http://www.opensourceforu.com/2010/10/debugging-linux-kernel-with-debugfs/

[30] Procfs, Sysfs, and Similar Kernel Interfaces. [Online] http://people.ee.ethz.ch/ arkeller/linux/multi/kernel_user_space_howto-2.html

[31] mount(8) - Linux man page. [Online] http://linux.die.net/man/8/mount