Driver development On Linux Platform for SPIRIT1

Submitted for partial fulfilment of the Degree

of

Bachelor of Technology

(Electronics and Communication Engineering)

Submitted By:

Ravi Kumar

130103059

Submitted To:

S.Mukherjee

Training Coordinator

ECE Department

Department of Electronics & Communication Engineering

Sharda University

Greater Noida

Abstract

SPIRIT1 a narrow band ultra-low power RF transceiver, intended for RF wireless applications in the sub-1 GHz band. It is designed to operate in both the license-free ISM and SRD frequency bands at 433,868 and 920 MHz, but can also be programmed to operate at other additional frequencies in the 430-470 MHz, 860-940 MHz bands. It comes under the category of Linux-WPAN. Linux-WPAN is managed by the Alexander Aring, Pengutronix. This technology is fit for exchanging the data over a short distance communications and also have many benefits over the existing system such as Bluetooth and Linux-WLAN.

Linux, a Unix-like and mostly POSIX compliant, open source operating system kernel is also the most used kernel in operating systems varying from PCs, mobiles, SOCs, etc.

This project report documents in detail the integration of SPIRIT, A ultra-low power transceiver by STMicroelectronics with the Linux Kernel. A detailed account is presented of the study of SPIRIT1 protocol Stack, Linux device model, and all other things that contributed to the final product i.e; a Linux device driver for SPIRIT1.

I

Acknowledgement

I, student of Sharda University, Greater Noida, have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. I would like to extend my sincere thanks to all of them.

The author is highly grateful to Dr. R.M. Mehra, Sharda University, Greater Noida for providing him with the opportunity to carry out his Six Months Internship at STMicroelectronics, Greater Noida.

The constant guidance and encouragement received from Dr. Pallavi gupta, ECE Co-ordinator, Sharda University, has been of great help in carrying out the training program and is duly acknowledged.

The author would like to whole heartedly thank Mr. Saurabh Rawat and Mr. Raunaque Quaiser for their constant guidance and direction. Without their help and motivation, it would not have been possible for this internship to go as smoothly and rewardingly as it did.

It is my absolute honor to place on record my best regards and deepest sense of gratitude to Mr. Manoj Kumar and all the mentors for their judicious and precious guidance and inspiration both of which were instrumental in growth as an engineer during my internship.

Finally, I would thank my fellow interns who have always uplifted my spirit and assisted in so many ways to my project.

Ravi Kumar

LIST OF FIGURES

INTRODUCTION TO ORGANIZATION

Figure 1.1: STMicroelectronics Pvt. Ltd., Greater Noida

STMicroelectronics is a world leader in providing the semiconductor solutions that make a positive contribution to people's lives, today and into the future.

Offering one of the industry's broadest product portfolios, ST serves customers across the spectrum of electronics applications with innovative semiconductor solutions for Smart Driving and the Internet of Things. By getting more from technology to get more from life, ST stands for life.augmented.

- Among the world's largest sermicondutor companies.

- A leading Integrated Device Manufacturer serving all electronics segments.

- A leading technology innovator (approximately 8,300 people working in R&D, 15,000 owned patents in 9,000 patent families and 500 new patent filings).

- Delivering solutions that are key to Smart Driving and Internet of Things.

- Rich, balanced portfolio (ASICs, Application-Specific Standard Products and Multi-Segment Products).

- A pioneer and visionary leader in sustainibility.

- Corporate Headquarters: Geneva, Switzerland.

- President and CEO: Carlo Bozotti.

- 2015 revenue: $6.90 billion.

- Approximately 43,200 employees.

- Over 75 sales & marketing offices in 35 counteries.

- 11 main manufacturing sites.

- Public since 1994 - traded on New York Stock Exchange (NYSE: STM), Euronext Paris, and Borsa Italiana.

- Created as SGS-THOMPSON Microelectronics in June 1987, from merger of SGS Microelettronica (Italy) and Thomson Semiconducteurs (France).

- Renamed STMicroelectronics in May 1998.

CHAPTER 2

INTRODUCTION TO PROJECT

The S2-LP is a very low-power RF transceiver, intended for RF wireless applications in the sub-1 GHz band. It is designed to operate both in the license-free ISM and SRD frequency bands at 169, 315, 433, 868, and 915 MHz, but can also be programmed to operate at other additional frequencies in the 300-348 MHz, 387-470 MHz, and 779-956 MHz bands. The air data rate is programmable from 1 to 500 kbps.

2.1 Overview

S2-LP follows the IEEE802.15.4 protocols. IEEE standard 802.15.4 intends to offer the fundamental lower network layers of a type of wireless personal area network (WPAN) which focuses on low-cost, low-speed ubiquitous communication between devices. It can be contrasted with other approaches, such as Wi-fi, which offer more bandwidth and require more power. The emphasis is on very low cost communication of nearby devices with little to no underlying infrastructure, intending to exploit this to lower power consumption even more. The basic framework conceives a 10-meter communications range with a transfer rate of 250 kbit/s. Tradeoffs are possible to favor more radically embedded devices with even lower power requirements, through the definition of not one, but several physical layers. Lower transfer rates of 20 and 40 kbit/s were initially defined, with the 100 kbit/s rate being added in the current revision. IEEE 802.25.4 includes the lower layer i.e; PHY and MAC layer. 6loWPAN sits on the top of the WPAN devices and act as the convergence layer to be used by the normal IPv6 kernel stack. 6LoWPAN transparently handles the fragmentation and defragmentation between the different maximum transmission units(MTUs) (127 vs 1280) as well as compressions.

The host system must be able to communicate with SPIRIT1 in order to set straight the relationships and interaction between various portions of the stack. These communications are of course handled by the Operating System. Furthermore for the OS to communicate with an external peripheral, device drivers are employed. And this is the whole idea of this project: creating a device driver for SPIRIT1 so that it can be used in conjunction with systems running on Linux based operating systems.

The device driver ensures that the peripheral is able to communicate with the host but the way this communication is to be conducted depends on what physical connection options are offered by the peripheral. In case of the SPIRIT1 the physical interface is SPI. Once the means of communication are fixed, there comes the question of deciding which part of the protocol is to reside on which side the host or the peripheral. It will be seen in later chapters as to how the whole stack has been fragmented in this project.

Linux is written in C and so are the device drivers for Linux.

2.2   Existing system

WPAN has been a prominent technology in networking and has been the backbone of many user applications. As such it only makes sense that an operating system incorporates enough attention to WPAN as a technology. And this is exactly the case with Linux. It was realized early on that WPAN is going to be a major player and hence efforts were made to have it be an integral part of Linux. The software side of WPAN is official WPAN protocol stack for Linux i.e. Linux-WPAN which will be looked at in later sections of this report. The other major thing that then remains is the hardware: the SPIRIT1. Due to increase in number of interconnect devices these days there is a greater need of software infrastructure to support these devices .Since the underlying hardware is a RF Device, and Linux is the Freely available OS with a large base of community support it is dire necessity to develop a Linux RF driver for the STs Sub-Ghz hardware SPIRIT1 a S2-LP.

This is of course possible because Linux, the OS has a stack support for SPI, so once a hardware is detected then all that remains is to bridge this stack to the device using some standard protocol. All of this is taken care by the device driver for that particular device. It is simple, in order for a device to be supported, there has to be a device driver for it, and in this section. I describe some of the widely used WPAN devices that are supported by Linux at the time this writing:

like Android, Meego ,Moblin ,RT-Linux and to larger base of Interconnect devices running Linux Like routers, network hubs, IoT devices and wearables. at86rfxxx, mrf24j40, cc2520.

2.2.1   ATMEL at86rf230

The AT86RF230 is a low-power 2.4 GHz radio transceiver which is developed by the ATMEL. Atmel Corporation is an American-based designer and manufacturer of semiconductors, founded in 1984. The company focuses on embedded systems built around microcontrollers. at86rf230 especially designed for ZigBee/IEEE 802.15.4 applications. The AT86RF230 follows the SPI for the communication with the OS. All RF-critical components except the antenna, crystal and decoupling capacitors are integrated on-chip. This single-chip radio transceiver provides a complete radio transceiver interface between the antenna and the microcontroller. It comprises the analog radio transceiver and the digital demodulation including time and frequency synchronization, and data buffering. In June 2014 they provide the software support for their device in Linux. The low power consumption of the device make it highly popular for the Linux OS and after that atmel developed many version of this transceiver(under this series at86rfxx).

Figure 2.1: atmel transceiver

2.2.2 Microchip mrf24j40

The MRF24J40 is an IEEE 802.15.4 Standard compliant 2.4 GHz RF transceiver. It is the product of Microchip. Microchip Technology is an American manufacturer of microcontroller, memory and analog semiconductors. Its products include microcontrollers (PIC micro, PIC32) Serial EEPROM devices, Serial SRAM devices, radio frequency (RF) devices. It integrates the PHY and MAC functionality in a single chip solution. The MRF24J40 creates a low-cost, low-power, low data rate (250 or 625 kbps) Wireless Personal Area Network (WPAN) device. The MRF24J40 interfaces to many popular Microchip PIC microcontrollers via a 4-wire serial SPI interface, interrupt, wake and Reset pins.

The MRF24J40 is compatible with Microchip's ZigBee, MiWi and MiWi P2P software stacks. It works in various mode but the IEEE 802.15.4 feature is supported only in normal mode and each TX FIFO has a specific purpose depending on if the mrf24j40 is configured for beacon and non-beacon-enabled mode. Microchip offers this transceiver in various OS.

Figure 2.2: Microchip transceiver

2.2.3 Texas CC2520

The CC2520 is TI's second generation ZigBee IEEE 802.15.4 RF transceiver for the 2.4 GHz unlicensed ISM band. Texas Instruments Inc. (TI) is an American technology company that designs and manufactures semiconductors, which it sells to electronics designers and manufacturers globally headquartered in Dallas, Texas, United States, TI is one of the top ten semiconductor companies worldwide, based on sales volume. This chip enables industrial grade applications by offering state-of-the-art selectivity/co-existence, excellent link budget, operation up to 125C and low voltage operation. CC2520 has 6 GPIO pins that can be individually configured as inputs, outputs and activate pull-up resistors. There are other solutions by Texas for WPAN as well and they work well with Linux as the driver is there for them.
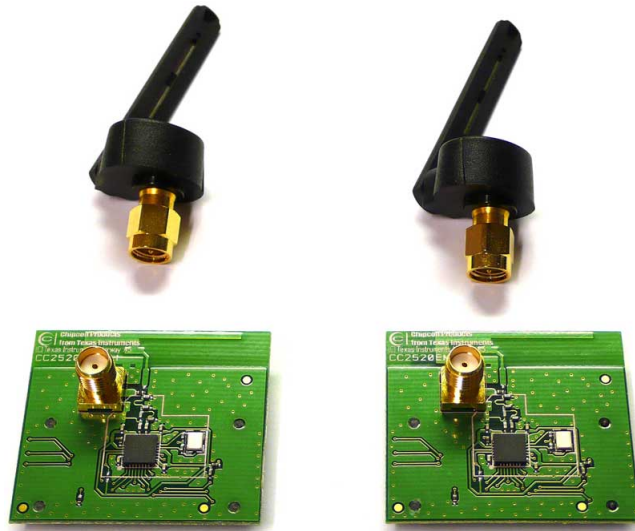
Figure 2.3: Texas transceiver

2.3 Requirement Analysis

There are many WPAN adapters out there that have well documented and tested support in Linux based operating systems. Linux being an open source kernel is always inviting people to look into its internal working and stack flow in order to contribute in it.

STMicroelectronics excels at grabbing newer opportunities and always be ready with solutions that are on par or even transcends other solutions in the market in a wide range of technologies. The WPAN technology is no different. ST has its fair share of WPAN modules. But thing to be considered here is that even though ST has a wide range of WPAN solutions, none has Linux support.

So the major requirement of this project was to take a WPAN transceiver by ST and add its support to Linux. And again Linux being open source makes the job much simpler. The module

that was considered is the S2-LP transceiver which is SPI based. So the whole requirement of this project can be summed up as adding support (writing a device driver) in Linux (because it is open source and one of the most popular OS kernels in the world) for the WPAN module offered by STMicroelectronic i.e. SPIRIT1 and S2-LP.

In order to fulfill the above stated requirement it is essential to look into both sides of the coin: Linux driver development and how is the WPAN stack being represented there; and then SPIRIT1 and what portion of the stack is on it and how much of it is needed in order to work with Linux. As S2-LP follows the IEEE 802.1.4 protocols. 802.15.4 covers the lower layer the physical layer and the medium acess control but the good news here is that in Linux already there is a support of 802.15.4 so we have to write the driver for our device and then registered that device on the IEEE 802.15.4 platform, for that we have to we have to look into the physical layer i.e. RF layer. Another important point to consider is that how the connection is to be made between the Host (running on Linux) and S2-LP. S2-LP is based on Serial Peripheral Interface, and in Linux already there is the support for Serial peripheral interface so that is exactly what is required to be used for the connections.

2.4 Feasbility Study

As stated in the requirement analysis section, there are two things that are needed to be considerd here:

2.4.1 Does SPIRIT1 and S2-LP support what WPAN offers?

The answer is yes. WPAN follows the IEEE802.15.4. These are the specification from the IEEE's 802.15 working group, which tackles wireless "personal area networks", or WPANs. The standard defines a physical-layer and media access control specification, meaning that it covers the frequencies and negotiation options of the connection, but not the protocols that run above it. The communication between between the two devices is to be held through the 6loWPAN these are also the standard developed by the IETF.

The 6LoWPAN concept originated from the idea that the Internet Protocol could and should be applied even to the smallest devices and in Linux already there is a support for the 6loWPAN because many transceiver from other manufacturer stated early follows the 6loWPAN for communications. We will discussed later that how the frame format for transmission of Ipv6 packets as well as well the formation Ipv6 link-local addresses and statelessly auto configured data on the top of IEE802.15.4 networks. Since Ipv6 support of packet size much larger than the largest IEEE802.15.4 frame size, an adaption layer is defined.

2.4.2 Does Linux supports what SPIRIT1 used to communicate?

SPIRIT1 is based on SPI. Any form of communication between the host and SPIRIT1 is to be had through the Serial Peripheral Interface. There are many SPI based driver in Linux for the WPAN. There are also many network drivers that allow the communication between the host and the controller through SPI. So following the same footsteps it should be possible to send commands from the host to the controller using SPI.

Considering the above things, it seems feasible to write a driver for SPIRIT1 in Linux. The driver shall essentially act as a bridge between SPIRIT1 protocol stack residing partly in the user space and partly in the kernel space of Linux and SPIRIT1.

2.5 Objectives of the Project

- To setup a SPIRIT1 and S2-LP driver interface for SPI.

- To make a entry for SPIRIT1 in sysfs.

- To map the IEEE 802.15.4 commands with the S2-LP.

- To interface the SPI Commands from SPIRIT1 to host.

3.1 Wireless Personal Area Network

WPAN **(wireless personal area network)** is a personal area network- a network for interconnecting devices centered around an individual person's workspace in which the connections are wireless. Typically, a wireless personal area network uses some technology that permits communication within about 10 meters - in other words, a very short range.

Wireless PAN is based on the standard **IEEE 802.15.4**. A WPAN could serve to interconnect all the ordinary computing and communicating devices that many people have on their desk or carry with them today; or it could serve a more specialized purpose such as allowing the surgeon and other team members to communicate during an operation. There are also many other existing standard such as the IEEE802.11 which is the (wireless local area network) WLAN so why there is a need to developed the new standards so the answer is simple that it requires less power, lesser bandwidth, has low penetration.

3.1.1 Limitations

WPAN has been designed with some specific fundamentals that happen to serve in a lot of application fields. But this does not means that it is not apt other kind of applications. Most of its limitations are due to the following two fundamental fact

1. The maximum therotical throughput is i**500Kbps** which is reduces by several factors to only **5-10KBps** in most cases.

2. The operating range is generally **2-5 meters**. This can be extended to **30 meters** or so but that will of course demand higher strain on the battery life of the devices. This is the defining thing about the BLE technology that the devices are mostly passive in nature and hence it becomes very important for them to have large battery life. In some cases it is seen that the devices battery can actually outlast other hardware in the device itself.

3. The loss of the data in WLAN goes upto **60** which is much higher than that of the WPAN ahich have only **30** data loss.

4. Frankly speaking, there is no much benefits of Wireless personal area network over wireless local area network **WLAN**the WLAN consumes more power because it transmits the data for a longer distance and have the throughput upto 10Mbps. WPAN useful for the short distance communication and low data transmission.

3.1.2   WPAN Architecture

3.1.2.1   Application layer

The upper layer containing most of the application logic, that is used to transfer the data. 6loW-PAN is the mostly used upper layer.

3.1.2.2   Host

It Contains all the lower layer protocols like PHY and MAC layer in WPAN IEEE802.15.4 acts as the host.

3.1.2.3   Controller

Linux acts as the controller for WPAN. We will discuseed all these layer one by one so first we start with host because it acts on the ground means it contains the RF layer.

3.1.3   IEEE 802.15.4 Network Topologies

Network topology defines the way in which different devices (under different roles) interact with one another to exchange data through transceiver. These topologies are defined under the specification and are maintained under the IEEE guidelines.
Depending on the application requirements, an IEEE 802.15.4 LR-WPAN operates in either of two topologies:

1. Star topology.

2. Peer to Peer topology.

Let's take a look ate each of them individually.

3.1.3.1   Star Topology

In the star topology, the communication is established between devices and a single central controller, called the PAN coordinator. A device typically has some associated application and is either the initiation point or the termination point for network communications. A PAN coordinator can also have a specific application, but it can be used to initiate, terminate, or route communication around the network. The PAN coordinator is the primary controller of the PAN All devices operating on a network of either topology have unique addresses referred to as extended addresses we will discussed it later that what is extended address. A device will use either the extended address for direct communication within the PAN or the short address that was allocated by the PAN coordinator when the device associated. The PAN coordinator will often be mains

powered, while the devices will most likely be battery powered. Applications that benefit from a star topology include home automation, personal computer (PC) peripherals, games, and personal health care.
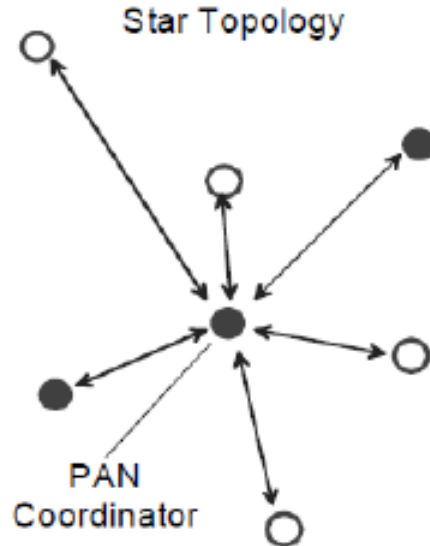
Figure 3.1: Star topology

3.1.3.2   Peer to Peer Topology

The peer-to-peer topology also has a PAN coordinator; however, it differs from the star topology in that any device is able to communicate with any other device as long as they are in range of one another. Peer-to-peer topology allows more complex network formations to be implemented, such as mesh networking topology. Applications such as industrial control and monitoring, wireless sensor networks, asset and inventory tracking, intelligent agriculture, and security would benefit from such a network topology. A peer-to-peer network allows multiple hops to route messages from any device to any other device on the network. Such functions can be added at the higher layer, but they are not part of this standard. Each independent PAN selects a unique identifier. This PAN identifier allows communication between devices within a network using short addresses and enables transmissions between devices across independent networks. The mechanism by which identifiers are chosen is outside the scope of this standard.
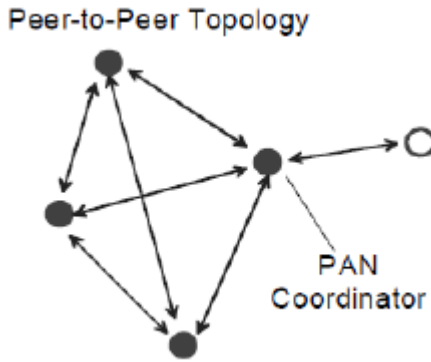
Figure 3.2: Peer to Peer topology

3.1.4 Network formation

3.1.4.1 Star network formation

After an FFD is activated, it can establish its own network and become the PAN coordinator. All-star networks operate independently from all other star networks currently in operation. This is achieved by choosing a PAN identifier that is not currently used by any other network within the radio communications range. Once the PAN identifier is chosen, the PAN coordinator allows other devices, potentially both FFDs and RFDs, to join its network.

3.1.4.2 Peer to Peer network formation

In a peer-to-peer topology, each device is capable of communicating with any other device within its radio communications range. One device is nominated as the PAN coordinator, for instance, by virtue of being the first device to communicate on the channel. Further network structures are constructed out of the peer-to peer topology, and it is possible to impose topological restrictions on the formation of the network.

An example of the use of the peer-to-peer communications topology is the **cluster tree**. The cluster tree network is a special case of a peer-to-peer network in which most devices are FFDs. An RFD connects to a cluster tree network as a leaf device at the end of a branch because RFDs do not allow other devices to associate. Any FFD is able to act as a coordinator and provide synchronization services to other devices or other coordinators. Only one of these coordinators is the overall PAN coordinator, potentially because it has greater computational resources than any other device in the PAN. The PAN coordinator forms the first cluster by choosing an unused PAN Identifier and broadcasting beacon frames to neighbouring devices.
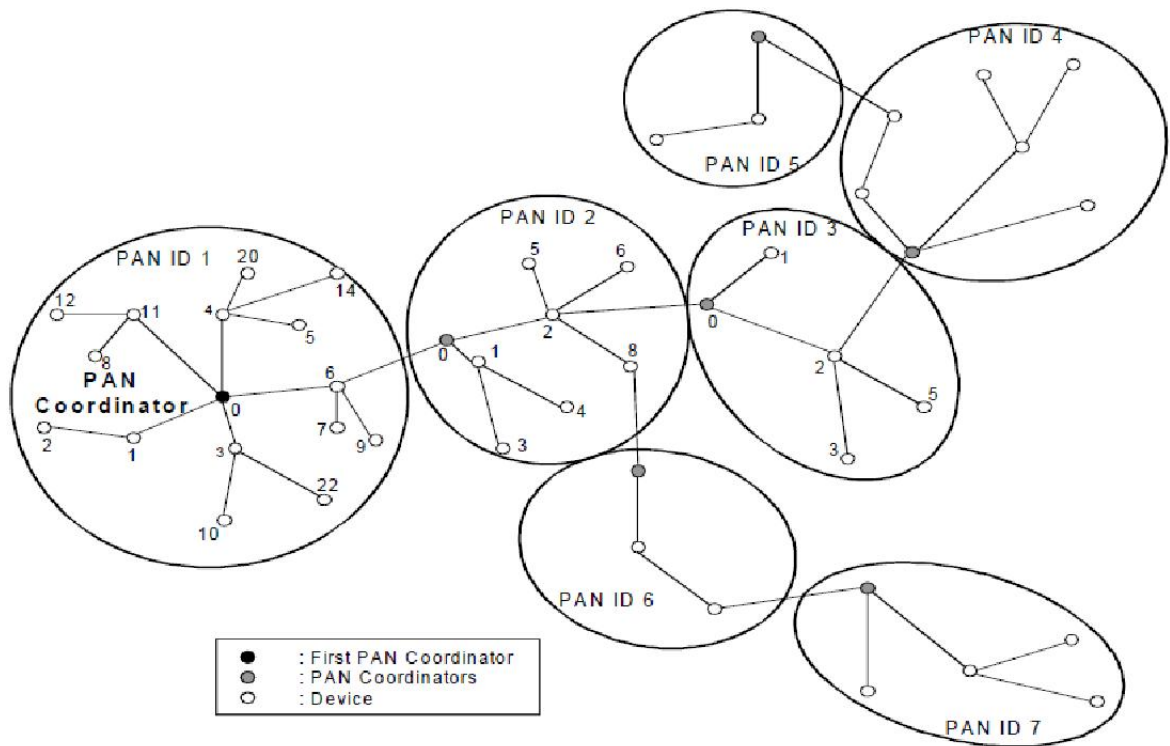
Figure 3.3: Cluster network

3.2   IEEE 802.15.4 Architecture

The IEEE 802.15.4 architecture is defined in terms of a number of blocks in order to simplify the standard. These blocks are called layers. Each layer is responsible for one part of the standard and offers services to the higher layers. As OSI have 7 layer but the IEEE 802.1.5.4 constitutes only the lower two layer

1. Physical layer(PHY)

2. Medium access control(MAC) layer

An LR-WPAN device comprises at least one PHY, which contains the radio frequency (RF) transceiver along with its low-level control mechanism, and a MAC sublayer that provides access to the physical channel for all types of transfer.

The upper layers, shown in figure, consist of a network layer, which provides network configuration, manipulation, and message routing, and an application layer, which provides the intended function of the device. The upper layer covered by the 6loWPAN. We will discussed the upper layers function in the coming chapter.
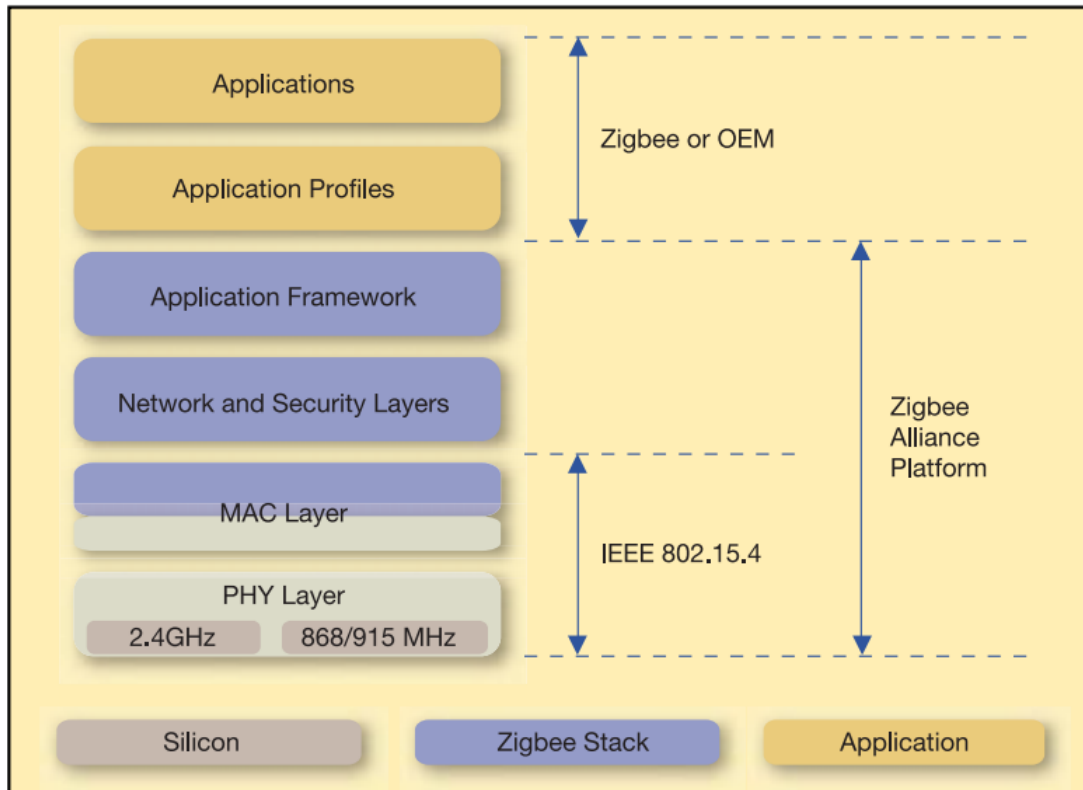
Figure 3.4: IEEE 802.15.4

3.2.1 Physical layer

The physical layer is the first layer of the Open System Interconnection Model (OSI Model). The physical layer deals with bit-level transmission between different devices and supports electrical or mechanical interfaces connecting to the physical medium for synchronized communication.
The PHY is responsible for the following tasks:

1. Activation and deactivation of the radio transceiver

2. Energy detection (ED) within the current channel

3. Link quality indicator (LQI) for received packets

4. Clear channel assessment (CCA) for carrier sense multiple access with collision avoidance**(CSMA/CA**

5. Channel frequency selection

6. Data transmission and reception

7. Precision ranging for ultra-wide band**(UWB)**PHY's

The PHYs defined in this standard are:

O-QPSK PHY:

Direct sequence spread spectrum (DSSS) PHY employing offset quadrature phase shift keying (O-QPSK) modulation, operating in the 780 MHz bands, 868 MHz, 915 MHz, and 2450 MHz.

BPSK PHY:

DSSS PHY employing binary phase-shift keying (BPSK) modulation, operating in the 868 MHz, 915 MHz, and 950 MHz bands.

ASK PHY:

Parallel sequence spread spectrum (PSSS) PHY employing amplitude shift keying.

ASK and BPSK modulation, operating in the 868 MHz and 915 MHz bands.

CSS PHY:

Chirp spread spectrum (CSS) employing differential quadrature phase-shift keying.

DQPSK:

DQPSK modulation, operating in the 2450 MHz band.

UWB PHY:

Combined burst position modulation (BPM) and BPSK modulation, operating in the sub-gigahertz and 310 GHz bands.

MPSK PHY:

M-ary phase-shift keying (MPSK) modulation, operating in the 780 MHz.

GFSK PHY:

Gaussian frequency-shift keying (GFSK), operating in the 950 MHz band.

3.2.2   Operating frequency range

A compliant device shall operate in one or several frequency bands using the modulation and spreading formats. Devices shall start in the PHY mode in which they are instructed to start. If the device is capable of operating in the 868 MHz or 915 MHz bands using one of the optional PHYs and, it shall be able to switch dynamically between the optional PHY and the mandatory BPSK PHY in that band when instructed to do so. If the 950 MHz band is supported, then at least one of the 950 MHz band PHYs shall be implemented.

3.2.3   Medium Acess Control

The Media Access Control Layer is one of two sublayers that make up the Data Link Layer of the OSI model sublayer handles all access to the physical radio channel and is responsible for the following tasks:

1. Generating network beacons if the device is a coordinator

2. Synchronizing to network beacons

3. Supporting PAN association and disassociation

4. Supporting device security

5. Employing the CSMA-CA mechanism for channel access

6. Handling and maintaining the GTS mechanism

7. Providing a reliable link between two peer MAC entities

There are two device types:

Full-function device (FFD)

Reduced-function device (RFD)

The FFD may operate in three modes serving as a personal area network (PAN) coordinator, a coordinator, or a device. An RFD shall only operate as a device.

3.2.3.1   MAC Services

The MAC sublayer provides an interface between the next higher layer and the PHY. The MAC sublayer conceptually includes a management entity called the mac layer management activity (MLME). This entity provides the service interfaces through which layer managementons may be invoked. The MLME is also responsible for maintaining a database of managed objects pertaining to the MAC sublayer. This database is referred to as the MAC sublayer PIB.
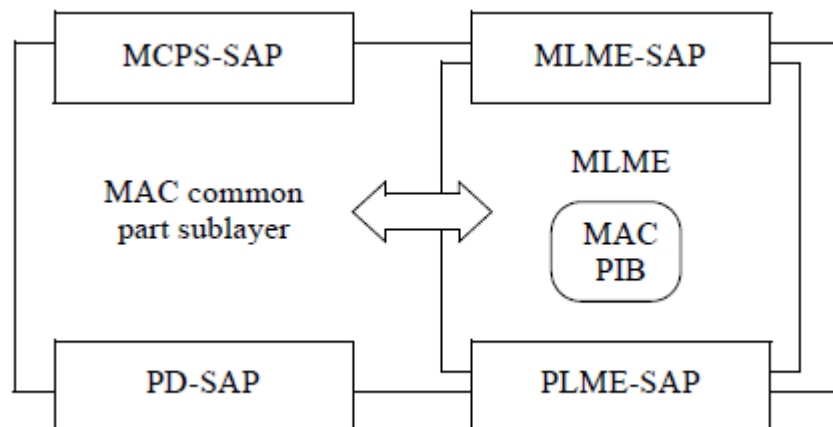
Figure 3.5: MAC Sublayer reference model

3.2.4 Security

The MAC sublayer is responsible for providing security services on specified incoming and outgoing frames when requested to do so by the higher layers. The 802.15.4 standard supports the following security services:

1. Data confidentiality

2. Data authenticity

3. Replay protection

A device may optionally implement security. A device that does not implement security shall not provide a mechanism for the MAC sublayer to perform any cryptographic transformation on incoming and outgoing frames nor require any PIB attributes associated with security. A device that implements security shall provide a mechanism for the MAC sublayer to provide cryptographic transformations on incoming and outgoing frames using information in the PIB attributes associated with security only if the macSecurityEnabled attribute is set to TRUE.

3.2.5 MAC layer advertising and scanning

The advertising packets are used to either broadcast data for applications or to discover slaves and to connect to them. The advertising of packets is done aftmake a entry for S2-LP in sysfs.er certain intervals and if during this interval some device is scanning then that device will be able to receive the advertising packet.

The advertising packets are used to either broadcast data for applications or to discover slaves and to connect to them. The advertising of packets is done after make a entry for S2-LP in sysfs after certain intervals and if during this interval some device is scanning then that device will be able to receive the advertising packet.

The specification defines the two types of scanning

3.2.5.1 Passive Scanning

The scanner simply listens for advertising packets, and the advertiser is never aware of the fact that one or more packets were actually received by a scanner.

3.2.5.2 Active Scanning

The scanner issues s Scan Request packet after receiving an advertising packet. The advertiser receives it and responds with a Scan Response packet. This additional packet doubles the effective

payload that the advertiser is able to send to the scanner, but it is important to note that this does not provide a means for the scanner to send any user data at all to the advertiser.

Before commencing an active or passive scan, the MAC sublayer shall store the value of macPANId and then set it to 0xffff for the duration of the scan this enables the receive filter to accept all beacons. On completion of the scan, the MAC sublayer shall restore the value of macPANId to the value stored before the scan began.

3.3   Low Wireless Personal Area Network

The 6LoWPAN group has defined encapsulation and header compression mechanisms that allow IPv6 packets to be sent and received over IEEE 802.15.4 based networks. IPv4 and IPv6 are the work horses for data delivery for local-area networks, metropolitan area networks, and wide-area networks such as the Internet. Likewise, IEEE 802.15.4 devices provide sensing communicationability in the wireless domain. The inherent natures of the two networks though, are different.

3.3.1   Transmission of IPv6 Packets over IEEE 802.15.4 Networks

The 6loWPAN sits over the IEEE802.15.4 packets, which acts as the convergence layer for the packet transmission.
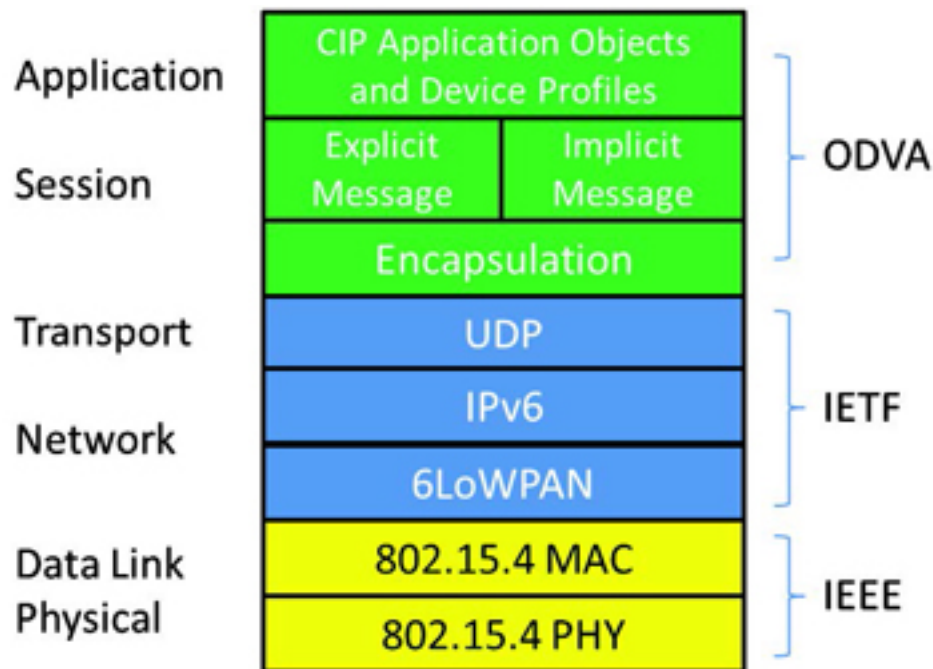
Figure 3.6: 6 LoWPAN over IEEE

3.3.2   IEEE 802.15.4 Mode over IPv6

IEEE 802.15.4 defines four types of frames

1. Beacon frames

2. MAC command frames

3. Acknowledgement frames

4. Data frames

IPv6 packets MUST be carried on data frames. Data frames may optionally request that they be acknowledged. IEEE 802.15.4 networks can either be nonbeacon-enabled or beacon-enabled. The latter is an optional mode in which devices are synchronized by a so-called coordinator's beacons. This allows the use of superframes within which a contention-free Guaranteed Time Service (GTS) is possible. In nonbeacon-enabled networks, data frames (including those carrying IPv6 packets) are sent via the contention-based channel access method of unslotted CSMA/CA. In nonbeacon-enabled networks, beacons are not used for synchronization. However, they are still useful for link-layer device discovery to aid in association and disassociation events.

3.3.3   Addressing Modes

IEEE 802.15.4 defines several addressing modes, it allows the use of either:
1. IEEE 64-bit extended addresses or (after an association event)
2. 16-bit addresses unique within the PAN.
6loWPAN supports both the addresses. Short addresses being transient in nature, a word of caution is in order. Since they are doled out by the PAN coordinator function during an association event, their validity and uniqueness is limited by the lifetime of that association.

3.3.4   Maximum transmission Unit

The MTU size for IPv6 packets over IEEE 802.15.4 is 1280 octets. However, a full IPv6 packet does not fit in an IEEE 802.15.4 frame. 802.15.4 protocol data units have different sizes depending on how much overhead is present. Starting from a maximum physical layer packet size of 127 octets (aMaxPHYPacketSize) and a maximum frame overhead of 25 (aMaxFrameOverhead), the resultant maximum frame size at the media access control layer is 102 octets. Link-layer security imposes further overhead, which in the maximum case (21 octets of overhead in the AES-CCM-128 case, versus 9 and 13 for AES-CCM-32 and AES-CCM-64, respectively) leaves only 81 octets available. This is obviously far below the minimum IPv6 packet size of 1280 octets. Furthermore,

since the IPv6 header is 40 octets long, this leaves only 41 octets for upper-layer protocols, like UDP. The latter uses 8 octets in the header which leaves only 33 octets for application data. Additionally, there is a need for a fragmentation and reassembly layer, which will use even more octets.

1. The adaptation layer must be provided to comply with the IPv6 requirements of a minimum MTU. However, it is expected that

   - Most applications of IEEE 802.15.4 will not use such large packets.

   - Small application payloads in conjunction with the proper header compression will produce packets that fit within a single IEEE 802.15.4 frame. The justification for this adaptation layer is not just for IPv6 compliance, as it is quite likely that the packet sizes produced by certain application exchanges (e.g., configuration or provisioning) may require a small number of fragments.

2. Even though the above space calculation shows the worst-case scenario, it does point out the fact that header compression is compelling to the point of almost being unavoidable. Since we expect that most (if not all) applications of IP over IEEE 802.15.4 will make use of header compression.
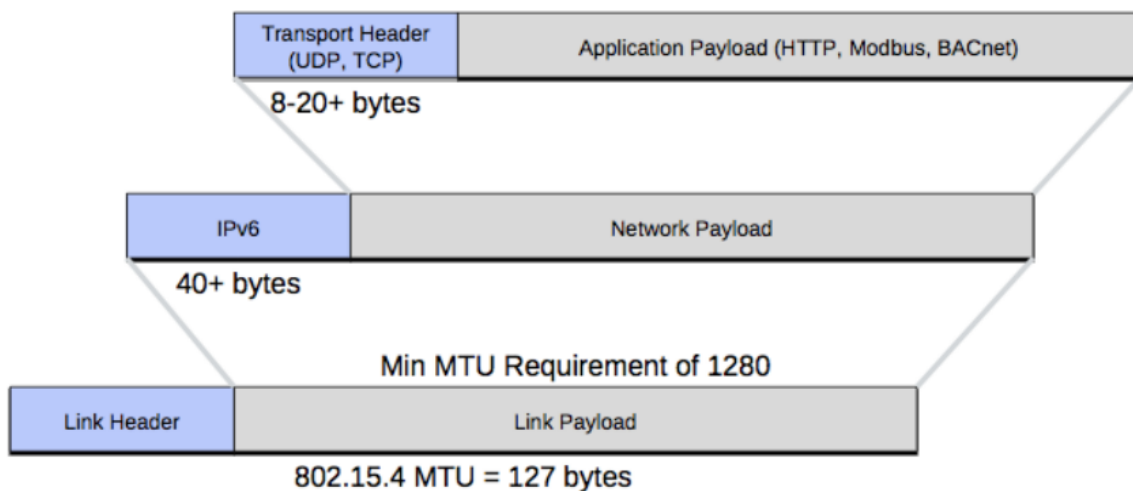
Figure 3.7: Fragmentation and defragmentation

3.3.5   Security of IPv6

The method of derivation of Interface Identifiers from EUI-64 MAC addresses is intended to preserve global uniqueness when possible. However, there is no protection from duplication through accident or forgery. Neighbor Discovery in IEEE 802.15.4 links may be susceptible to threats. Mesh routing is expected to be common in IEEE 802.15.4 networks. This implies additional threats due to ad hoc routing. IEEE 802.15.4 provides some capability for link-layer security. Users are urged to make use of such provisions if at all possible and practical. Doing so will alleviate the threats stated above.

A sizeable portion of IEEE 802.15.4 devices is expected to always communicate within their PAN (i.e., within their link, in IPv6 terms). In response to cost and power consumption considerations, and in keeping with the IEEE 802.15.4 model of "Reduced Function Devices" (RFDs), these devices will typically implement the minimum set of features necessary. Accordingly, security for such devices may rely quite strongly on the mechanisms defined at the link layer by IEEE 802.15.4. The latter, however, only defines the Advanced Encryption Standard (AES) modes for authentication or encryption of IEEE 802.15.4 frames, and does not, in particular, specify key management (presumably group oriented). Other issues to address in real deployments relate to secure configuration and management. Whereas such a complete picture is out of the scope of this document, it is imperative that IEEE 802.15.4 networks be deployed with such considerations in mind. Of course, it is also expected that some IEEE 802.15.4 devices (the so-called "Full Function Devices", or "FFDs") will implement coordination or integration functions. These may communicate regularly with off-link IPv6 peers (in addition to the more common on-link exchanges). Such IPv6 devices are expected to secure their end-to-end communications with the usual mechanisms (e.g., IPsec, TLS, etc).

3.4   WPAN The official Linux S2-LP stack

WPANis the official Linux personal area protocol stack. It is an Open Source project distributed under the GNU General Public License (GPL).

Figure 3.8: WPAN the official S2-LP stack of Linux

3.4.1 History

The linux-WPAN project was started in 2008. Initially, it was started with the name of linux-zigbee but this name is little confusing because there is no zigbee protocol at all and Linux is an open source wheras zigbee is managed by zigbee community. The lower layer consist of the PHY and MAC layer as we discussed above and the transport layer consist of the Zigbee protocol. Mainly Community slowly took over and the new open standard was formed to avoid confusion with the name linux-WPAN and the new maintainer were Alexander Aring and Pengutronix. Now there are many higher layer protocols which make the use of IEEE 802.15.4 profiles.

These are discussed below:

1. Zigbee

   - Zigbee Alliance's mesh networking protocol but its not available under the GPL licensing as we discussed above.

   - MiWi Mesh and MiWi P2P

   - Microchip's proprietary mesh and P2P protocols. This protocol is only compatible with the Microchips component and they have their own standard.

   - 6LoWPAN

   - IPv6 over 802.15.4.

   - WirelessHART

   - Its mainly used for Industrial Automation purpose because it required the more power and data sent by this protocol is of higher bits.
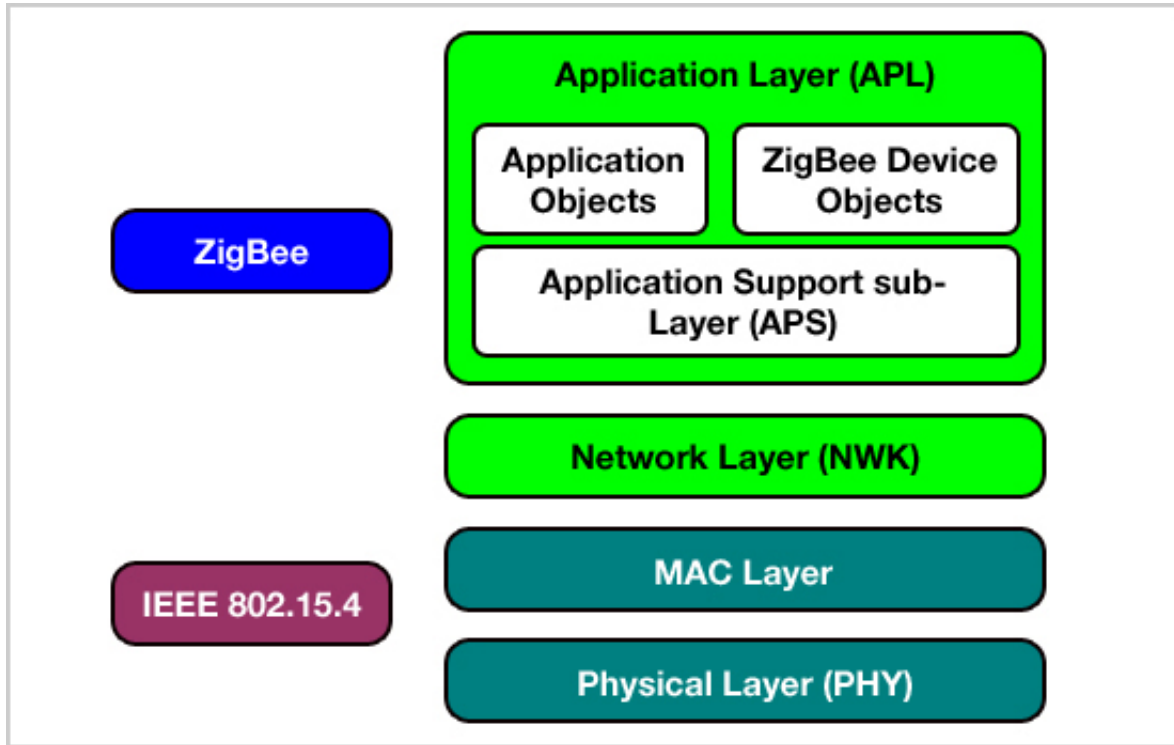
Figure 3.9: Linux-WPAN

So we have only one option left i.e. IPv6 over 802.15.4.

But many of the protocols of the linux-WPAN is taken from the linux-WLAN. If we compare both of the frameworks then.

WPAN	Description	WLAN
WLAN	Default interfacing naming	WPAN
station	Default interfacing type registration	node
iw	command framework	iWPAN
nl80211	Netlink kernel framework	nl802154
cfg80211	Soft and Hard-MAC Interface	cfg802154

Table 3.1: WLAN and WPAN framework

3.5 S2-LP

The S2-LP is a narrow band ultra-low power RF transceiver, intended for RF wireless applications in the sub-1 GHz band. It is designed to operate in both the license-free ISM and SRD frequency bands at 433,868 and 920 MHz, but can also be programmed to operate at other additional frequencies in the 430-470 MHz, 860-940 MHz bands. The S2-LP supports different modulation schemes: 2(G)FSK, 4(G)FSK, OOK and ASK. The air data rate is programmable from 0.3 to 500 kbps. The S2-LP can be used in systems with channel spacing of 12.5/25 kHz, complying with the EN 300 220 standard and satisfying the latest FCC narrow banding mandate. The S2-LP shows a RF link budget higher than 140dB for long communication range and meets the regulatory requirements applicable in territories worldwide, including Europe, Japan, China and USA. The S2-LP integrates a configurable baseband modem with proprietary fully programmable packet format allowing also:

1. IEEE 802.15.4g full hardware packet supporting whitening, CRC, FEC and dual SYNC word detection.

2. Wireless M-Bus standard compliance packet format (all performance classes). In order to reduce the overall system power consumption and increase the communication reliability, the S2-LP provides an embedded programmable automatic packet acknowledgment, automatic packet retransmission, CSMA/CA engine, low duty cycle protocol, RX sniff mode and timeout protocol. The S2-LP fully supports antenna diversity with an integrated antenna switching control algorithm. Transmitted/received data bytes are buffered in two different 128 bytes FIFOs (TX FIFO and RX FIFO), accessible via SPI interface for host processing. In addition the reduction of external components enables easily and fast integrate the S2-LP on products to enable a short time to market.

3.5.1 Block diagram of S2-LP

The receiver architecture is low-IF conversion, the received RF signal is amplified by a two-stage lownoise amplifier (LNA) and down-converted in quadrature (I and Q) to the intermediate frequency (IF). LNA and IF amplifiers make up the RX front-end (RXFE) and have programmable gain. The transmitter part of the S2-LP is based on direct synthesis of the RF frequency. The power amplifier (PA) input is the LO generated by the RF synthesizer, while the output level can be configured between -30 dBm and +14 dBm (+16 dBm in boost mode), at pin level with 0.5 dB steps. The data to be transmitted can be provided by an external MCU either through the 128-byte TX FIFO writable via SPI, or directly using a programmable GPIO pin.

The S2-LP supports frequency hopping, TX/RX and antenna diversity switch control, extending the link range and improving performance. The S2-LP has a very efficient power management

(PM) system. An integrated switched mode power supply (SMPS) regulator allows operation from a battery voltage ranging from +1.8 V to +3.6 V, and with power conversion efficiency of 90percent.
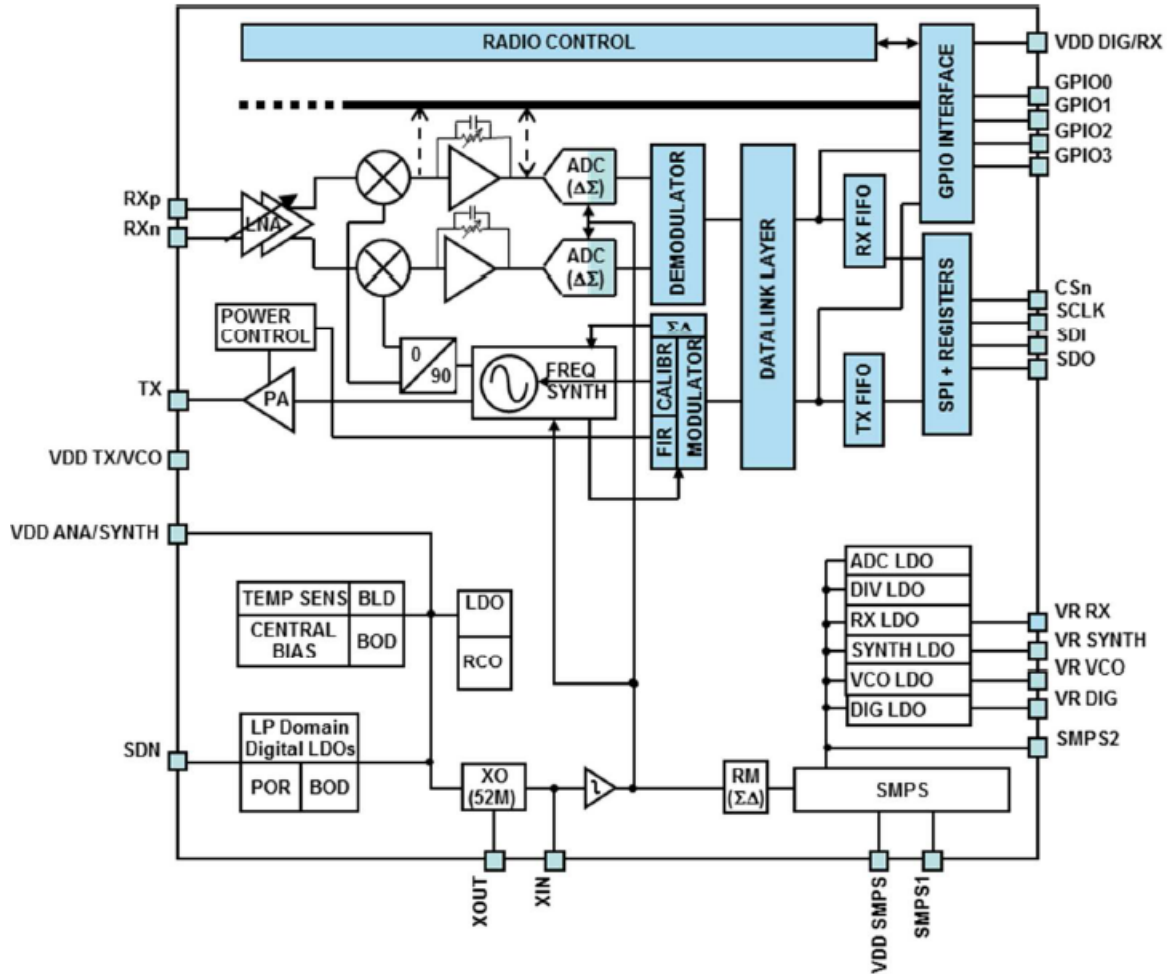
Figure 3.10: Block diagram of S2-LP

3.5.2 SPIRIT1

The SPIRIT1 is a very low-power RF transceiver, intended for RF wireless applications in the sub-1 GHz band. It is designed to operate both in the license-free ISM and SRD frequency bands at 169, 315, 433, 868, and 915 MHz, but can also be programmed to operate at other additional frequencies in the 300-348 MHz, 387-470 MHz, and 779-956 MHz bands. The air data rate is programmable from 1 to 500 kbps, and the SPIRIT1 can be used in systems with channel spacing of 12.5/25 kHz, complying with the EN 300 220 standard. It uses a very small number of discrete external components and integrates a configurable baseband modem, which supports data management, modulation, and demodulation.

The data management handles the data in the proprietary fully programmable packet format also

allows the M-Bus standard compliance format (all performance classes). However, the SPIRIT1 can perform cyclic redundancy checks on the data as well as FEC encoding/decoding on the packets. The SPIRIT1 provides an optional automatic acknowledgement, retransmission, and timeout protocol engine in order to reduce overall system costs by handling all the high-speed link layer operations. Moreover, the SPIRIT1 supports an embedded CSMA/CA engine. An AES 128-bit encryption co-processor is available for secure data transfer. The SPIRIT1 fully supports antenna diversity with an integrated antenna switching control algorithm. The SPIRIT1 supports different modulation schemes: 2-FSK, GFSK, OOK, ASK, and MSK. Transmitted/received data bytes are buffered in two different three-level FIFOs (TX FIFO and RX FIFO), accessible via the SPI interface for host processing.

3.5.3   SPIRIT1/S2-LP Protocol Stack
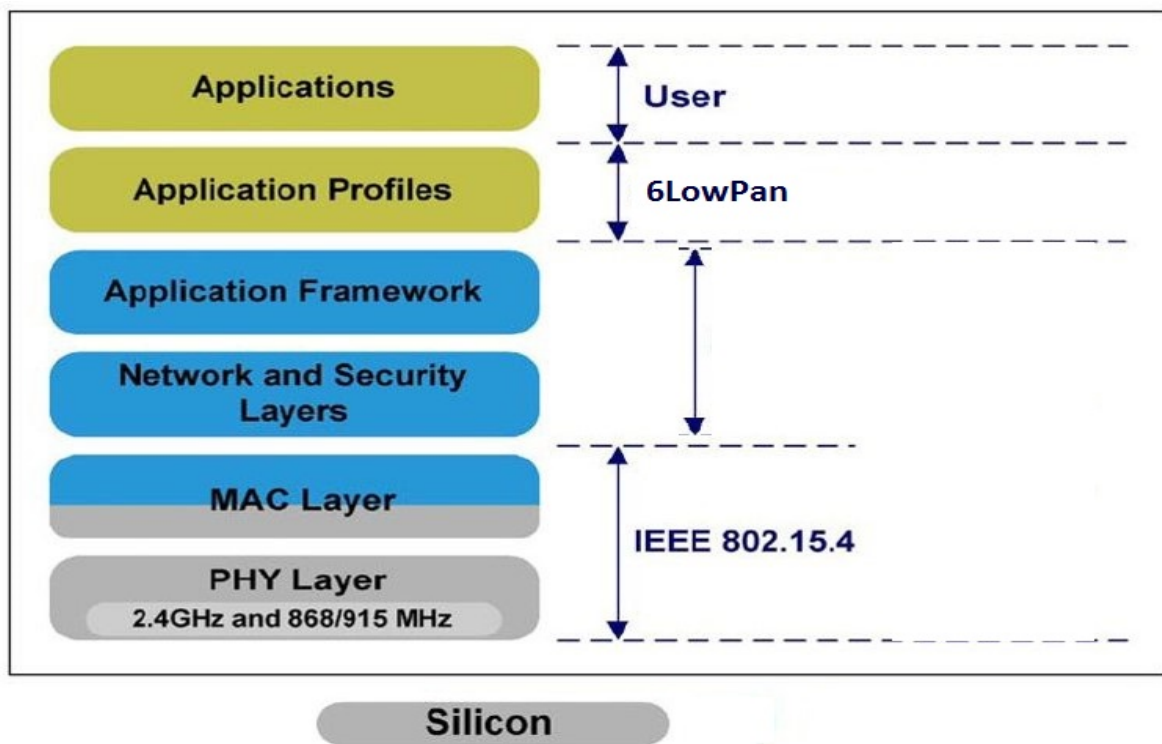
Figure 3.11: SPIRIT1/S2-LP protocol stack

Architecture of SPIRIT1 is similar to S2-LP except it does not follows IEEE 802.15.4 stack which is provided by the Linux kernel. If we talk about the protocols and Block Diagram then both are same but the only difference is that S2-LP has 128 bits of FIFO register whereas SPIRIT1 has 96 bits of FIFO register.

3.5.4 Pin Diagram of S2-LP
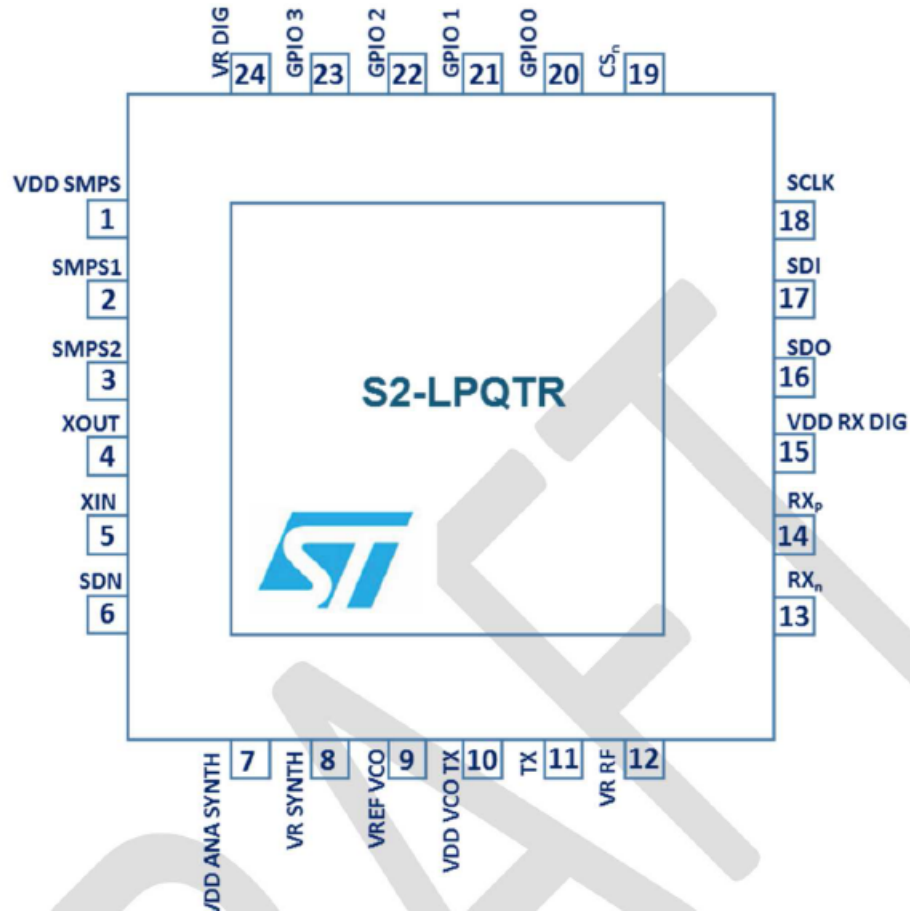
Figure 3.12: Pin configuration of S2-LP

3.5.5 Pin Description

In this Report we mainly focused for the SPIRIT1 driver writing because I wrote the Driver for SPIRIT1, as we discussed in the previous section that the SPIRIT1 and S2-LP architecture are same except the IEEE protocols.

N	Pin Name	Pin type	Description
1	VDD SMPS	POWER	
1	VDD SMPS	POWER	1.8 to 3.6 V analog power supply for SMPS only
2	SMPS1	Analog out	1.1 to 1.8 V SMPS regulator output to be externally filtered
3	SMPS2	Analog in	1.1 to 1.8 SMSP voltage input after LC filtering applied to SMPS1 output
4	XOUT	Analog out	Crystal oscillator output. connect to an external crystal or leave floating if driving the XIN pin with an external clock
5	XIN	Analog in	Crystal oscillator input. Connect to an external oscillator or to an external clock source
6	SDN	Digital in	Shutdown input pin. SDN should be = '0' in all modes but '1' in SHUTDOWN mode
7	VDD ANA/SYNTH	Power	1.8 to 3.6 V
8	VR SYNTH	Analog in/out	1.2 V SYNTH-LDO output for decoupling
9	VREF VCO	Analog out	1.2 V VCO-LDO for decoupling
10	VDD VCO/TX	Power	1.8 V to 3.6 V power supply
11	TX	RF output	RF output signal
12	VR RF	Analog in/out	1.2 V RX-LDO for decoupling
13	RXn	RF in	Differential RF input signals for the LNA
14	RXp	RF in	Differential RF input signals for the LNA
15	VDD RX/DIG	Power	1.8 to 3.6 V power supply
16	SDO	Digital out	SPI slave data output
17	SDI	Digital In	SPI slave data input
18	SCLK	Digital in	SPI slave clock input
19	CSn	Digital in	SPI chip select
20	GPIO0	Digital I/O	General purpose I/O
21	GPIO1	Digital I/O	General purpose I/O
22	GPIO2	Digital I/O	General purpose I/O
23	GPIO3	Digital I/O	General purpose I/O
24	VR DIG	Analog in/out	1.2 V supply for decoupling

Table 3.2: SPIRIT1 pin description

3.5.6   Operating Modes

Several operating modes are defined for the SPIRIT1:

- Reset Mode

- Sleep Mode

- Standby Mode

- Ready Mode

- Lock Mode

- Shutdown Mode

The SPIRIT1 is provided with a built-in main controller which controls the switching between the two main operating modes: transmit (TX) and receive (RX). In shutdown condition (the SPIRIT1 can be switched on/off with the external pin SDN, all other functions/registers are available through the SPI interface and GPIOs), no internal supply is generated (in order to have minimum battery leakage), and hence, all stored data and configurations are lost. The GPIO and SPI ports during SHUTDOWN are in HiZ. From shutdown, the SPIRIT1 can be switched on from the SDN pin and goes into READY state, which is the default, where the reference signal from XO is available. From READY state, the SPIRIT1 can be moved to LOCK state to generate the high precision LO signal and/or TX or RX modes. Switching from RX to TX and vice versa can happen only by passing through the LOCK state. This operation is normally managed by radio control with a single user command (TX or RX). At the end of the operations above, the SPIRIT1 can return to its default state (READY) and can then be put into a sleeping condition (SLEEP state), having very low power consumption. If no timeout is required, the SPIRIT1 can be moved from READY to STANDBY state, which has the lowest possible current consumption while retaining FIFO, status and configuration registers. To manage the transitions towards and between these operating modes, the controller works as a state machine, whose state switching is driven by SPI commands.

The SPIRIT1 radio control has three stable states (READY, STANDBY, LOCK) which may be defined stable, and they are accessed by the specific commands (respectively READY, STANDBY, and LOCKRX/LOCKTX), which can be left only if any other command is used. All other states are transient, which means that, in a typical configuration, the controller remains in those states, at most for any timeout timer duration. Also the READY and LOCK states behave as transients when they are not directly accessed with the specific commands (for example, when LOCK is temporarily used before reaching the TX or RX states).
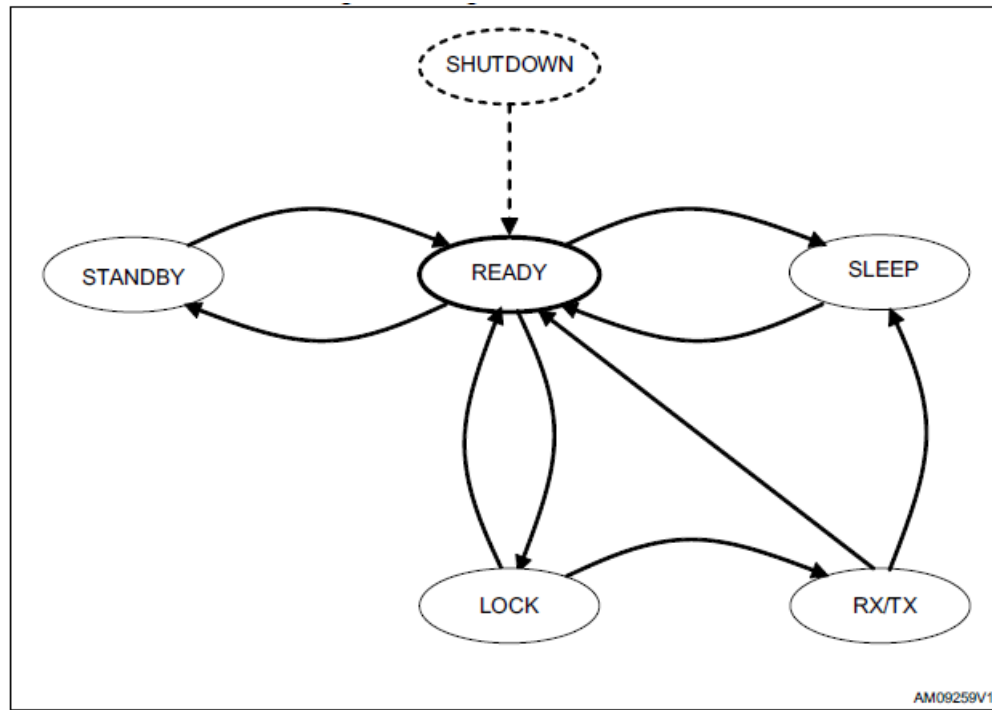
Figure 3.13: Operating modes of SPIRIT1

STATE[6:0][1]	State/mode	Digital LDO	SPI	Xtal	RF Synth.	Wake-up timer	Response time to[2]	
							TX	RX
-	SHUTDOWN	OFF (register contents lost)	Off	Off	Off	Off	NA	NA
0x40	STANDBY	ON (FIFO and register contents retained)	On	Off	Off	Off	125 µs	125 µs
0x36	SLEEP		On	Off	Off	On	125 µs	125 µs
0x03	READY (Default)		On	On	Off	Don't care	50 µs	50 µs
0x0F	LOCK		On	On	On	Don't care	NA	NA
0x33	RX		On	On	On	Don't care	15 µs[3]	NA
0x5f	TX		On	On	On	Don't care	NA	15 µs[3]

Figure 3.14: Spirit opearting modes summary

DEVELOPMENT AND IMPLEMENTATION

4.1 The Linux Kernel

The Linux kernel is a monolithic Unix-like computer operating system kernel. The Linux operating system is based on it and deployed on both traditional computer systems such as personal computers, servers and small system on chip computers; usually in the form of Linux distributions, and on various embedded devices such as routers, wireless access points, PBXes, set-top boxes, FTA receivers, smart TVs, PVRs and NAS appliances. The Android operating system for tablet computers, smartphones and smart-watches is also based atop the Linux kernel.

The Linux kernel was conceived and created in 1991 by Linux Torvalds for his personal computer and with no cross-platform intentions, but has since expanded to support to huge array of computer architecture, many more than other operating systems or kernels. Linux rapidly attracted developers and users who adopted it as the kernel for other free software projects, notably the GNU Operating System. The Linux kernel has received contributions from nearly 12,000 programmers from more than 1,200 companies, including some of the largest software and hardware vendors.

4.1.1 The Linux Device Driver Model

Linux today supports more hardware devices than any other operating system in the history of the world. It does this using a development model significantly different from the familiar Windows device driver model. The Linux development process continues to evolve to better support the needs of Independent Hardware Vendors (IHVs), distributions, and other members of the community, and the advantages of the Linux model are increasing with time. While Linux will not provide a stable source or binary interface for driver developers, IHVs should familiarize themselves with a number of useful projects, many sponsored by the Linux Foundation, that ease driver development, including the Hardware NDA program, the Linux Drivers Project, and the Driver Backport Workgroup. When IHVs engage with the Linux community, they almost invariably find that the Linux driver model provides significant benefits that lower their costs while producing better drivers.

4.1.1.1 Overview

A fundamental purpose for operating systems (OSes) is to serve as an abstraction layer between applications and hardware to enable interoperability. An IHV wants their hardware to be able to make use of all the relevant features of anOS, and anOSwants to take full advantage of the hardware it's running on. Since both theOSand the hardware tend to add and to rearrange features over

time, it is a dynamic interaction. What everybody wants is for the hardware to Just Work without hassles or support calls.

Today, Linux works with more devices than any otherOSin the history of the world.

4.1.1.2   Driver Model

The Linux driver model is different. For users, the goal is to provide the Just Works experience. The Linux model is that IHVs get the source code for their driver accepted into the mainline kernel. This entails a public peer review process to ensure that the driver code is of sufficient quality and does not have obvious bugs or security risks.Linux has neither a stable binary driver ABI nor a stable source-code driver Application Programming Interface (API). That is, there is no guarantee that an interface provided in one version of the kernel will be available in the next version, and portions of the ABI andAPIchange in every kernel release.

By contrast, the Linux kernel does provide a stable user-space interface for Linux applications. These applications essentially have a contract with the Linux kernel that the user-space binary interfaces they rely on will continue to work consistently over time. That's why a pre-compiled Linux application can run correctly on multiple distributions and multiple versions. The underlying implementation of the user-space binary interfaces can and does change, but even an application compiled for pre-1.0 Linux will run correctly on the latest kernel. This is the opposite of device drivers, which have no guarantee whatsoever that any interface they rely on, whether binary or source, will remain consistent between versions of the kernel.

Counterintuitive though it might be from a proprietary viewpoint, this lack of internal kernel interface stability is preferable because both the kernel code and all of the drivers relying on it are open source. In fact, driver code is an integral part of the Linux operating system, not a second-class add-on. Once a driver is accepted into the mainline kernel, it will be maintained over time as internal kernel interfaces change. That is, when a subsystem maintainer accepts a patch to make an incompatible change to a kernel interface, that patch will simultaneously upgrade every driver that relies on the interface. And, new drivers and any upgrades to them automatically flow downstream from the mainline kernel to all Linux distributions.

The key strength of this approach from the user's viewpoint is that, in happy contrast with proprietary operating systems like Windows Vista, once a device is working on a given version of Linux, that support continues through all future versions. (Devices are generally only removed when they have become so rare that no users can be found.) In Linux, hardware support only gets better; it never gets worse.

From the IHV's point of view, the big benefit is that an IHV's driver is maintained over time by the community, meaning that other people fix, tune, and add features to the driver. When internal kernel interfaces change in each newOSrelease, IHVs don't need to write and release a new driver; their driver is upgraded automatically. Obsolete interfaces can be deprecated and removed rather

than being maintained indefinitely. Common subsystems can be factored out of drivers, enabling leaner, less buggy device drivers while adding more functionality for all hardware. This improves the stability, security, and maturity of both theOSand the driver.

In addition, the Linux model enables cross-architecture driver support nearly for free. Even when an IHV only tests their driver on one chip architecture, interested developers ensure that the driver works with every architecture that Linux supports, which is more than any otherOSin history. The strength of this approach has been especially apparent over the last decade as many chip architectures have moved from the 32 to 64 bits. Nearly all Linux drivers were quickly updated to support these newer architectures, while driver support for 64-bit Windows Vista even on the highest volume x86 architecture remains extremely poor today.

The biggest hurdle of the Linux driver model for some IHVs is the need to open source their driver code, which a small (but thankfully dwindling) number have been reluctant to do. Also, once a driver is accepted into the mainline, it can take up to 18 months to be deployed into an enterprise distro. It has not until now been convenient to backport the driver to existing distros, but that is improving with the Driver Backport Workgroup.

Having hardware reliably supported by Linux means getting the driver accepted into the mainline kernel. Supporting an out-of-mainline open source driver creates significant, never-ending support costs for the IHV, as new versions constantly need to be released as the kernelAPIchanges. Supporting an out-of-mainline binary driver means even bigger, never-ending support costs for the IHV, and directly contradicts the recent statement by a large number of kernel developers that binary drivers are undesirable.

The Linux driver model is different from the Windows model many IHVs are used to. But it is a consistent and compelling approach, and has been successful at supporting nearly the entire universe of computer hardware. Moreover, the vast majority of all IHVs have adapted to Linux and have thriving businesses that work with the Linux driver development model.

4.1.2 Overview of SPI support in Linux

4.1.2.1 What is SPI?

The "Serial Peripheral Interface" (SPI) is a synchronous four wire serial link used to connect microcontrollers to sensors, memory, and peripherals. It's a simple "de facto" standard, not complicated enough to acquire a standardization body. SPI uses a master/slave configuration.

The three signal wires hold a clock (SCK, often on the order of 10 MHz), and parallel data lines with "Master Out, Slave In" (MOSI) or "Master In, Slave Out" (MISO) signals. (Other names are also used.) There are four clocking modes through which data is exchanged; mode-0 and mode-3 are most commonly used. Each clock cycle shifts data out and data in; the clock doesn't cycle except when there is a data bit to shift. Not all data bits are used though; not every protocol uses

those full duplex capabilities.

SPI masters use a fourth "chip select" line to activate a given SPI slave device, so those three signal wires may be connected to several chips in parallel. All SPI slaves support chipselects; they are usually active low signals, labeled nCSx for slave 'x' (e.g. nCS0). Some devices have other signals, often including an interrupt to the master.

Unlike serial busses like USB or SMBus, even low level protocols for SPI slave functions are usually not interoperable between vendors (except for commodities like SPI memory chips).

- SPI may be used for request/response style device protocols, as with touchscreen sensors and memory chips.

- It may also be used to stream data in either direction (half duplex), or both of them at the same time (full duplex).

- Some devices may use eight bit words. Others may use different word lengths, such as streams of 12-bit or 20-bit digital samples.

- Words are usually sent with their most significant bit (MSB) first, but sometimes the least significant bit (LSB) goes first instead.

- Sometimes SPI is used to daisy-chain devices, like shift registers.

In the same way, SPI slaves will only rarely support any kind of automatic discovery/enumeration protocol. The tree of slave devices accessible from a given SPI master will normally be set up manually, with configuration tables.

SPI is only one of the names used by such four-wire protocols, and most controllers have no problem handling "MicroWire" (think of it as half-duplex SPI, for request/response protocols), SSP ("Synchronous Serial Protocol"), PSP ("Programmable Serial Protocol"), and other related protocols.

Some chips eliminate a signal line by combining MOSI and MISO, and limiting themselves to half-duplex at the hardware level. In fact some SPI chips have this signal mode as a strapping option. These can be accessed using the same programming interface as SPI, but of course they won't handle full duplex transfers. You may find such chips described as using "three wire" signaling: SCK, data, nCSx. (That data line is sometimes called MOMI or SISO.)

Microcontrollers often support both master and slave sides of the SPI protocol. This document (and Linux) currently only supports the master side of SPI interactions.

4.1.2.2   Who uses SPI?

Linux developers using SPI are probably writing device drivers for embedded systems boards. SPI is used to control external chips, and it is also a protocol supported by every MMC or SD memory

card. (The older "DataFlash" cards, predating MMC cards but using the same connectors and card shape, support only SPI.) Some PC hardware uses SPI flash for BIOS code.

SPI slave chips range from digital/analog converters used for analog sensors and codecs, to memory, to peripherals like USB controllers or Ethernet adapters; and more.

Most systems using SPI will integrate a few devices on a mainboard. Some provide SPI links on expansion connectors; in cases where no dedicated SPI controller exists, GPIO pins can be used to create a low speed "bitbanging" adapter. Very few systems will "hotplug" an SPI controller; the reasons to use SPI focus on low cost and simple operation, and if dynamic reconfiguration is important, USB will often be a more appropriate low-pincount peripheral bus.

Many microcontrollers that can run Linux integrate one or more I/O interfaces with SPI modes. Given SPI support, they could use MMC or SD cards without needing a special purpose MMC, SD or SDIO controller.

4.1.2.3   The SPI programming interface

The linux/spi/spi.h header file includes kerneldoc, as does the main source code, and you should certainly read that chapter of the kernel API document. This is just an overview, so you get the big picture before those details.

SPI requests always go into I/O queues. Requests for a given SPI device are always executed in FIFO order, and complete asynchronously through completion callbacks. There are also some simple synchronous wrappers for those calls, including ones for common transaction types like writing a command and then reading its response.

There are two types of SPI driver, here called:

Controller drivers ... controllers may be built into System-On-Chip processors, and often support both Master and Slave roles. These drivers touch hardware registers and may use DMA. Or they can be PIO bitbangers, needing just GPIO pins.

Protocol drivers ... these pass messages through the controller driver to communicate with a Slave or Master device on the other side of an SPI link.

So for example one protocol driver might talk to the MTD layer to export data to filesystems stored on SPI flash like DataFlash; and others might control audio interfaces, present touchscreen sensors as input interfaces, or monitor temperature and voltage levels during industrial processing. And those might all be sharing the same controller driver.

A "struct spi_device" encapsulates the master-side interface between those two types of driver. At this writing, Linux has no slave side programming interface.

There is a minimal core of SPI programming interfaces, focussing on using the driver model to connect controller and protocol drivers using device tables provided by board specific initialization code. SPI shows up in sysfs in several locations:

- /sys/devices/.../CTLR ... physical node for a given SPI controller

- /sys/devices/.../CTLR/spiB.C... spi_device on bus "B", chipselect C, accessed through CTLR.

- /sys/bus/spi/devices/spiB.C ... symlink to that physical .../CTLR/spiB.C device

- /sys/devices/.../CTLR/spiB.C/modalias ... identifies the driver that should be used with this device (for hotplug/coldplug)

- /sys/bus/spi/drivers/D ... driver for one or more spi*.* devices

- /sys/class/spi_master/spiB ... symlink (or actual device node) to a logical node which could hold class related state for the controller managing bus "B". All spiB.* devices share one physical SPI bus segment, with SCLK, MOSI, and MISO.

Note that the actual location of the controller's class state depends on whether you enabled CONFIG_SYSFS_DEPRECATED or not. At this time, the only class-specific state is the bus number ("B" in "spiB"), so those /sys/class entries are only useful to quickly identify busses.

4.2 The C Programming Language

C is the language of the Linux kernel. Almost all of the kernel and related applications are written in C. C is of course the language of development in this project.
C is a general-purpose programming language with features economy of expression, modern flow control and data structures, and a rich set of operators. C is not a "very high level", language, nor a big one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages. C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11, by Dennis Ritchie. The operating system, the C compiler, and essentially all UNIX applications programs (including all of the software used to prepare this book) are written in C. Production compilers also exist for several other machines, including the IBM System/370, the Honeywell 6000, and the Interdata 8/32. C is not tied to any particular hardware or system, however, and it is easy to write programs that will run without change on any machine that supports C.
The C language has a very simple structure and hence is much easily converted into machine code without any overhead that may be faced when using other languages. That is why almost all of the kernel development and device drivers are done in C language only even after the introduction of so many other versatile languages like C++.

4.3   Development Boards

In order to develop and test the driver code as well as to interface the SPIRIT1 chip to the host via SPI, there was need for a system-on-board computer. The host of course needed to run on a Linux kernel, so the following two were the obvious choices

4.3.1   Raspberry Pi 2

Several generations of Raspberry Pis have been released. The first generation (Raspberry Pi 1 Model B) was released in February 2012. It was followed by a simpler and inexpensive model Model A. In 2014 the foundation released a board with an improved design in Raspberry Pi 1 Model B+. The model laid the current "mainline" form-factor. Improved A+ and B+ models were released a year later. A cut down "compute" model was released in April 2014, and a Raspberry Pi Zero with smaller size and limited input/output (I/O) and general-purpose input/output (GPIO) abilities was released in November 2015 for US$5. The Raspberry Pi 2 which added more RAM was released in February 2015. Raspberry Pi 3 Model B released in February 2016 is bundled with on-board WiFi and Bluetooth. As of 2016, Raspberry Pi 3 Model B is the newest mainline Raspberry Pi. These boards are priced between US $2035.

All models feature a Broadcom system on a chip (SoC), which includes an ARM compatible central processing unit (CPU) and an on chip graphics processing unit (GPU, a VideoCore IV). CPU speed ranges from 700 MHz to 1.2 GHz for the Pi 3 and on board memory range from 256 MB to 1 GB RAM. Secure Digital (SD) cards are used to store the operating system and program memory in either the SDHC or MicroSDHC sizes. Most boards have between one and four USB slots, HDMI and composite video output, and a 3.5 mm phone jack for audio. Lower level output is provided by a number of GPIO pins which support common protocols like IC. The B-models have an 8P8C Ethernet port and the Pi 3 has on board Wi-Fi 802.11n and Bluetooth.

The Foundation provides Raspbian, a Debian-based Linux distribution for download, as well as third party Ubuntu, Windows 10 IOT Core, RISC OS, and specialised media center distributions. It promotes Python and Scratch as the main programming language, with support for many other languages. The default firmware is closed source, while an unofficial open source is available. The Raspberry Pi 3 model was used for preliminary development of this driver.

Figure 4.1: Raspberry Pi 2

4.3.2   BeagleBone Black

The BeagleBoard is a low-power open-source hardware single-board computer produced by Texas Instruments in association with Digi-Key and Newark element14. The BeagleBoard was also designed with open source software development in mind, and as a way of demonstrating the Texas Instrument's OMAP3530 system-on-a-chip. The board was developed by a small team of engineers as an educational board that could be used in colleges around the world to teach open source hardware and software capabilities. It is also sold to the public under the Creative Commons share-alike license. The board was designed using Cadence OrCAD for schematics and Cadence Allegro for PCB manufacturing; no simulation software was used.

The BeagleBone Black is the newest member of the BeagleBoard family. It is a lower-cost, high-expansion focused BeagleBoard using a low cost Sitara XAM3359AZCZ100 Cortex A8 ARM processor from Texas Instruments. It is similar to the Beaglebone,but with some features removed and some features added. The table below gives the high points on the differences between the BeagleBone and BeagleBone Black.
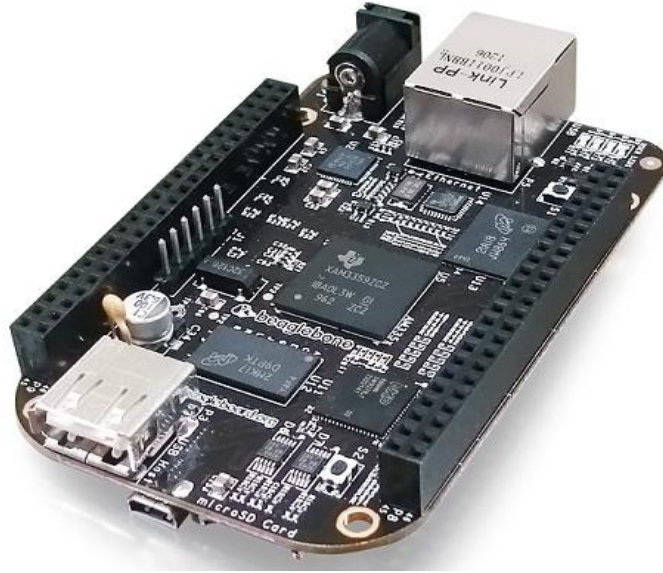
Figure 4.2: BeagleBone Black

4.4 Tools used

4.4.1 Buildroot

Buildroot is a set of Makefiles and patches that simplifies and automates the process of building a complete and bootable Linux environment for an embedded system, while using cross-compilation to allow building for multiple target platforms on a single Linux-based development system. Buildroot can automatically build the required cross-compilation toolchain, create a root file system, compile a Linux kernel image, and generate a boot loader for the targeted embedded system, or it can perform any independent combination of these steps. For example, an already installed cross-compilation toolchain can be used independently, while Buildroot only creates the root file system.

Buildroot is primarily intended to be used with small or embedded systems based on various computer architectures and instruction set architectures (ISAs), including x86, ARM, MIPS and PowerPC. Numerous architectures and their variants are supported; Buildroot also comes with default configurations for several off-the-shelf available embedded boards, such as Cubieboard, Raspberry Pi and SheevaPlug. Several third-party projects and products use Buildroot as the basis for their build systems, including the OpenWrt project that creates an embedded operating system, and firmware for the customer-premises equipment (CPE) used by the Google Fiber broadband service.

Multiple C standard libraries are supported as part of the toolchain, including the GNU C Library, uClibc and musl, as well as the C standard libraries that belong to various preconfigured development environments, such as those provided by Linaro. Buildroot's build configuration sys-

tem internally uses Kconfig, which provides features such as a menu-driven interface, handling of dependencies, and contextual help; Kconfig is also used by the Linux kernel for its source-level configuration. Buildroot is organized around numerous automatically downloaded packages, which contain the source code of various userspace applications, system utilities, and libraries. Root file system images, which are the final results, may be built using various file systems, including cramfs, JFFS2, romfs, SquashFS and UBIFS.

Buildroot is free and open-source software, maintained by Peter Korsgaard and licensed under version 2 or later of the GNU General Public License (GPL). The project started in 2001, with initial intentions to serve as a testbed for uClibc. New releases are made available every three months.
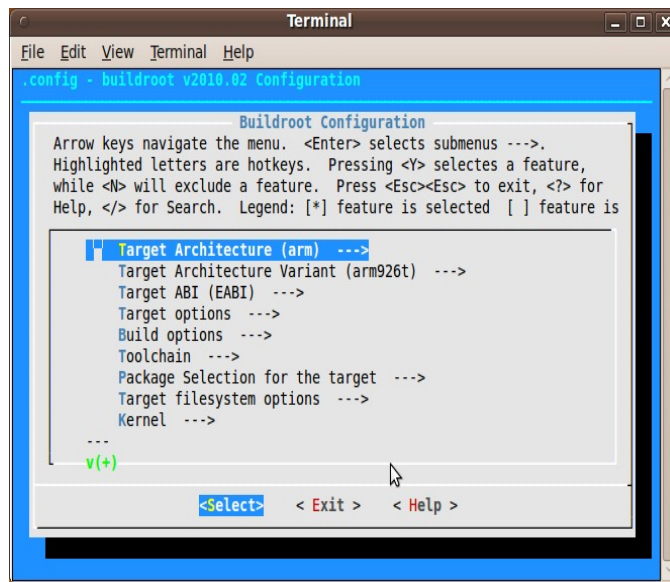

Figure 4.3: Buildroot

4.4.2 Cscope

While hacking around the Linux source code it becomes very difficult to keep track of the program flow or the variable structures in your head. There isnt any IDE to help out in source code browsing either. In such situation a tool is necessary that browses source code and is based on the terminal. This is where cscope comes into picture.

cscope is a programming tool which works in console mode, text-based interface, that allows computer programmers or software developers to search source code of the programming language C, with some support for C++ and Java. It is often used on very large projects to find source code, functions, declarations, definitions and regular expressions given a text string. cscope is free and released under a BSD license. The original developer of cscope is **Joe Steffen**.

The history of the tool goes back to the days of the PDP-11, but it is still used by developers who

are accustomed to using the vi or Vim editor or other text-based editors, instead of editors based on graphical user interfaces (GUI)s. The functions in cscope are available to varying degrees in modern graphical source editors. cscope is used in two phases. First a developer builds the cscope database. The developer can often use find or other Unix tools to get the list of filenames needed to index into a file called cscope.files. The developer then builds a database using the command cscope -b -q -k. The k flag is intended to build a database for an operating system or C library source code. It will not look in /usr/include. Second, the developer can now search those files using the command cscope -d. Often an index must be rebuilt whenever changes are made to files. In software development it is often very useful to be able to find the callers of a function because this is the way to understand how code works and what other parts of the program expect from a function. cscope can find the callers and callees of functions, but it is not a compiler and it does that by searching the text for keywords. This has the disadvantages that macros and duplicate symbol names can generate an unclear graph. There are other programs that can extract this information by parsing the source code or looking at the generated object files. cscope was created to search content within C files, but it can also be used (with some limits) for C++ and Java files.
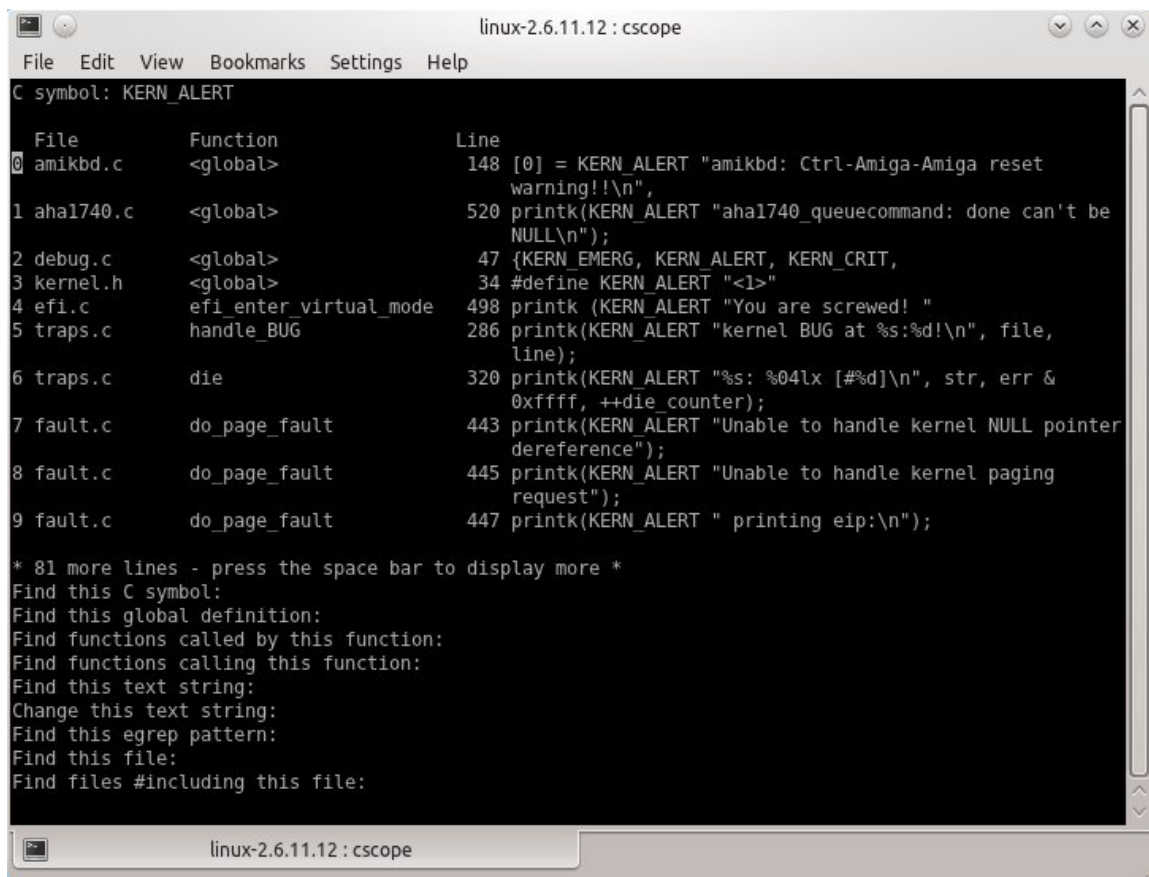

Figure 4.4: Source Browsing through Cscope

4.4.3   VIM

Vim (a contraction of Vi IMproved) is a clone of Bill Joy's vi text editor program for Unix. It was written by Bram Moolenaar based on source for a port of the Stevie editor to the Amiga and first released publicly in 1991. Vim is designed for use both from a command-line interface and as a standalone application in a graphical user interface. Vim is free and open source software and is released under a license that includes some charityware clauses, encouraging users who enjoy the software to consider donating to children in Uganda. The license is compatible with the GNU General Public License. Although Vim was originally released for the Amiga, Vim has since been developed to be cross-platform, supporting many other platforms. In 2006, it was voted the most popular editor amongst Linux Journal readers; in 2015 the Stack Overflow developer survey found it to be the third most popular text editor; and in 2016 the Stack Overflow developer survey found it to be the fourth most popular development environment. Although there are many editor available such as Nano. In Vim we have two modes Insert mode and command mode. Insert mode is started by pressing the I and then we insert the words whereas command mode is start by pressing the Esc key.

Some basic commands are :

- Moving through the text is usually possible with the arrow keys, but in the command mode try:

- h to move the cursor to the left

- l to move it to the right

- k to move up

- j to move down

- SHIFT-G will put the prompt at the end of the document.

4.4.4   Logic Analyzer

A logic analyzer is an electronic instrument that captures and displays multiple signals from a digital system or digital circuit. A logic analyzer may convert the captured data into timing diagrams, protocol decodes, state machine traces, assembly language, or may correlate assembly with source-level software. Logic Analyzers have advanced triggering capabilities, and are useful when a user needs to see the timing relationships between many signals in a digital system.

Figure 4.5: Logic Analyzer

4.4.5 Programming interface

4.4.5.1 Makefile for SPIRIT1

To compile the code of the SPIRIT1 as a module we need the Makefile which we already discussed in the previous section. Now we will see the format of the Makefile:

MODULES := spi.o

ARCH := arm

CROSS_COMPILE :=/home/zinga/Ravi/oldrasberry/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin/arm-linux-gnueabihf-

obj-m := $(MODULES) OUTDIR := /home/zinga/Ravi/oldrasberry/linux

MAKEARCH := $(MAKE) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE)

all: modules

modules:

$(MAKEARCH) -C $(OUTDIR) M=${shellpwd}modules

clean :

$(MAKEARCH) - C$(OUTDIR)M = ${shellpwd}clean

4.4.5.2   Linux Kernel SPI support for SPIRIT1

The "Serial Peripheral Interface" (SPI) is a synchronous four wire serial link used to connect micro-controllers to sensors, memory, and peripherals. It's a simple "de facto" standard, not complicated enough to acquire a standardization body. SPI uses a master/slave configuration. The three signal wires hold a clock (SCK, often on the order of 10 MHz), and parallel data lines with "Master Out, Slave In" (MOSI) or "Master In, Slave Out" (MISO) signals. (Other names are also used.) There are four clocking modes through which data is exchanged; mode-0 and mode-3 are most commonly used. Each clock cycle shifts data out and data in; the clock doesn't cycle except when there is a data bit to shift. Not all data bits are used though; not every protocol uses those full duplex capabilities.

SPI masters use a fourth "chip select" line to activate a given SPI slave device, so those three signal wires may be connected to several chips in parallel. All SPI slaves support chipselects; they are usually active low signals, labeled nCSx for slave 'x' (e.g. nCS0). Some devices have other signals, often including an interrupt to the master.
SPI is only one of the names used by such four-wire protocols, and most controllers have no problem handling "MicroWire" (think of it as half-duplex SPI, for request/response protocols), SSP ("Synchronous Serial Protocol"), PSP ("Programmable Serial Protocol"), and other related protocols.

4.4.5.2.1   Four modes in SPI   The four modes combine two mode bits:

- CPOL indicates the initial clock polarity. CPOL=0 means the clock starts low, so the first (leading) edge is rising, and the second (trailing) edge is falling. CPOL=1 means the clock starts high, so the first (leading) edge is falling.

- CPHA indicates the clock phase used to sample data; CPHA=0 says sample on the leading edge, CPHA=1 means the trailing edge.

- Since the signal needs to stablize before it's sampled, CPHA=0 implies that its data is written half a clock before the first clock edge. The chipselect may have made it become available. Chip specs won't always say "uses SPI mode X" in as many words, but their timing diagrams will make the CPOL and CPHA modes clear. In the SPI mode number, CPOL is the high order bit and CPHA is the low order bit. So when a chip's timing diagram shows the clock starting low (CPOL=0) and data stabilized for sampling during the trailing clock edge (CPHA=1), that's SPI mode 1.

  Note that the clock mode is relevant as soon as the chipselect goes active. So the master must set the clock to inactive before selecting a slave, and the slave can tell the chosen polarity by

sampling the clock level when its select line goes active. That's why many devices support for example both modes 0 and 3: they don't care about polarity, and always clock data in/out on rising clock edges.

4.4.5.2.2 Write the SPI master controller driver An SPI controller will probably be registered on the platform_bus; write a driver to bind to the device, whichever bus is involved. The main task of this type of driver is to provide an "spi_master". Use spi_alloc_master() to allocate the master, and spi_master_get_devdata() to get the driver-private data allocated for that device. struct spi_master *master;
struct CONTROLLER *c;
master = spi_alloc_master(dev, sizeof *c);
if (!master)
return -ENODEV;
c = spi_master_get_devdata(master);
The driver will initialize the fields of that spi_master, including the bus number (maybe the same as the platform device ID) and three methods used to interact with the SPI core and SPI protocol drivers. It will also initialize its own internal state.
If we need to remove your SPI controller driver, spi_unregister_master() will reverse the effect of spi_register_master().

4.4.5.2.3 SPI master methods master->setup(struct spi_device *spi)
This sets up the device clock rate, SPI mode, and word sizes. Drivers may change the defaults provided by board_info, and then call spi_setup(spi) to invoke this routine. It may sleep.
Unless each SPI slave has its own configuration registers, don't change them right away ... otherwise drivers could corrupt I/O that's in progress for other SPI devices.

- BUG ALERT: for some reason the first version of

- many spi_master drivers seems to get this wrong.

- When you code setup(), ASSUME that the controller is actively processing transfers for another device.

master->cleanup(struct spi_device *spi)
Controller driver may use spi_device.controller_state to hold state it dynamically associates with that device. If we do that, be sure to provide the cleanup() method to free that state.

4.4.5.2.4 Write the SPI protocol driver Most SPI drivers are currently kernel drivers, but there's also support for userspace drivers. Here we talk only about kernel drivers.

SPI protocol drivers somewhat resemble platform device drivers:

static struct spi_driver CHIP_driver = {

.driver = {

.name = "*CHIP*",

.owner = *THIS_MODULE*,

.pm = *CHIP_pm_ops*,

},

.probe = *CHIP_probe*,

.remove = *CHIP_remove*,

}.

Now compile the driver using the below command

make

and with the lsmod we are able to show the module of SPIRIT1.

After this we are able to see our device i.e. SPIRIT1 in the spi category.

cd /class/spi/

4.4.6   Transfer the data over SPI

By making the entry of our device in the SPI now we have to transfer the data over the SPI. For that we have to make the read and write function for the SPI.

- To read the data from SPI.
  int (*read) (struct stsp_dev *dev, u8 addr, int len, u8 *data);
  To read the data we have to give the address of the buffer, length of the data that has to be read and the data.

- To write the data from SPI.
  int (*write) (struct stsp_dev *dev, u8 addr, int len, char *data);
  To write the data over the SPI we have to follow the above function.
  In the stsp_dev function we have to give the device information.

Now to transfer the data over the the syntax is:

static const struct stsp_transfer_function stsp_spi_tf = {

.write = *stsp_spi_write*,

.read = *stsp_spi_read*,

}

After this we are able to send the data over SPI.

4.4.6.1 SYSFS programming Interface

sysfs is a ram-based filesystem initially based on ramfs. It provides a means to export kernel data structures, their attributes, and the linkages between them to userspace.

sysfs is always compiled in if CONFIG_SYSFS is defined. We can access it by doing:

mount -t sysfs sysfs /sys

For every kobject that is registered with the system, a directory is created for it in sysfs. That directory is created as a subdirectory of the kobject's parent, expressing internal object hierarchies to userspace. Top-level directories in sysfs represent the common ancestors of object hierarchies; i.e. the subsystems the objects belong to.

Sysfs internally stores a pointer to the kobject that implements a directory in the kernfs_node object associated with the directory. In the past this kobject pointer has been used by sysfs to do reference counting directly on the kobject whenever the file is opened or closed. With the current sysfs implementation the kobject reference count is only modified directly by the function sysfs_schedule_callback().

4.4.6.1.1 Attributes   Attributes can be exported for kobjects in the form of regular files in the filesystem. Sysfs forwards file I/O operations to methods defined for the attributes, providing a means to read and write kernel attributes.

Attributes should be ASCII text files, preferably with only one value per file. It is noted that it may not be efficient to contain only one value per file, so it is socially acceptable to express an array of values of the same type. An attribute definition is simply:

struct attribute {

$char * name$;

$struct module * owner$;

$umode\_t mode$;

};

int sysfs_create_file(struct kobject * kobj, const struct attribute * attr);

void sysfs_remove_file(struct kobject * kobj, const struct attribute * attr);

A bare attribute contains no means to read or write the value of the attribute.Subsystems are encouraged to define their own attribute structure and wrapper functions for adding and removing attributes for a specific object type.

struct device_attribute {

$struct attribute attr$;

$ssize\_t(*show)(struct device * dev, struct device\_attribute * attr, char * buf)$;

$ssize\_t(*store)(struct device * dev, struct device\_attribute * attr, const char * buf, size\_t count)$;

};

Now we have to define the device attributes.

define DEVICE_ATTR(_name, _mode, _show, _store)

struct device_attribute dev_attr_mode_name = _ATTR(_name, _mode, _show, _store)

4.4.7   Reading/writing Attribute data

To read or write attributes, show() or store() methods must be specified when declaring the attribute. The method types should be as simple as those defined for device attributes:

ssize_t (*show)(struct device *dev, struct device_attribute *attr, char *buf);

ssize_t (*store)(struct device *dev, struct device_attribute *attr, const char *buf, size_t count);

IOW, they should take only an object, an attribute, and a buffer as parameters.

sysfs allocates a buffer of size (PAGE_SIZE) and passes it to the method. Sysfs will call the method exactly once for each read or write.

The format of the implementation of the syfs for the SPIRIT1 is:

static ssize_t show_name(struct device *dev, struct device_attribute *attr, char *buf)

{

$return scnprintf(buf, PAGE\_SIZE, ``\%s'', dev\text{-}¿name);$

}

$static ssize\_t store\_name(struct device * dev, struct device\_attribute * attr, const char * buf, size\_t count)$

{

$snprintf(dev->name, sizeof(dev->name), ``\%.*s'', (int)min(count, sizeof(dev->name)-1), buf);$

$return count;$

}

$static DEVICE\_ATTR(name, S\_IRUGO, show\_name, store\_name);$

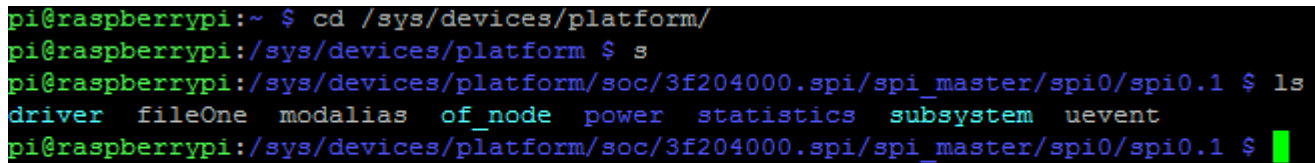The top level sysfs directory looks like:

- block

- bus

- class

- dev

- devices

- firmware

- net

- fs

- devices contains a filesystem representation of the device tree. It maps directly to the internal kernel device tree, which is a hierarchy of struct device.

- bus contains flat directory layout of the various bus types in the kernel. Each bus's directory contains two subdirectories:
  devices
  drivers

- devices contains a directory for each device driver that is loaded for devices on that particular bus (this assumes that drivers do not span multiple bus types).

By making the entry of our device in the sysfs we are able to see it in the sysfs directory under spi directory as shown in below figure:

cd /sys/devices/platform/spi

Using the above command we are able to see the sysfs file for SPIRIT1.

Figure 4.6: entry of SPIRIT1 in sysfs

To read the contents of the sysfs we have to change the mode using below command:

sudo chmod 666 -R fileOne

4.4.8 Initialize the hardware

After making the entry for the SPIRIT1 in sysfs we have to initialize the hardware i.e. SPIRIT1 on linux platform for that we have to make an entry of the hardware in the probe function. probe() will be called to make sure that the device exist and the functionality is fine.If device is not hot-pluggable, functionality of probe() can be put inside init() method.This will reduce driver's run time memory footprint. Probe() happens at the time of device boot or when device is connected.For a "platform" device the probe function is invoked when a platform device is registered and it's device name matches the name specified on the device driver.

To make an entry in the probe function the syntax is:

err = stsp_hw_init(dev);

Now we have to initialize the hardware.

static int stsp_hw_init(struct stsp_dev *dev)

{

$interr;$

$u8data;$

$err = dev- > tf- > read(dev, DEVICE\_INFO0\_VERSION, 1, data);$

$printk(``probingthefunction = ravi'');$

$staticintstsp\_hw\_init(structstsp\_dev * dev)$

{

$interr;$

$u8data;$

$err = dev- > tf- > read(dev, DEVICE\_INFO0\_VERSION, 1, data);$

$printk(``probingthefunction = ravi'');$

$if(err < 0)$

{

$dev\_err(dev- > dev, ``errorreadingtheversion'');$

$returnerr;$

}

$printk(``probbing = \%d'', err);$

$if(data! = DEVICE\_INFO0\_VERSION)$

{

$dev\_err(dev- > dev, ``deviceunknownx\%02x - 0x'', DEVICE\_INFO0\_VERSION, data);$
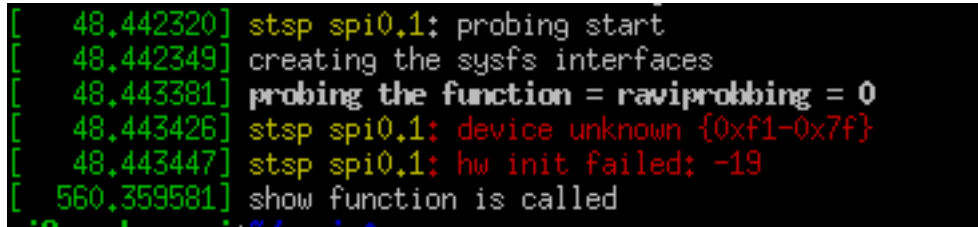
$return - ENODEV;$

}

$returnerr;$

}

After initializing the hardware and insert the modules using

sudo insmod spi.ko

and then checking the result running the below command

dmesg

we will see the entry of hardware as shown in figure below. I am getting this output because hardware is not connected to my RPI board we will see the result with the hardware connecting in the testing section. Here we have to check only whether the hardware entry is made or not in Linux.

Figure 4.7: Hardware initialization in Linux

4.4.9   Port the essential stuff

After initializing the hardware now our main focus is to write the essential stuff for the SPIRIT1. Essential stuff means Register, Interrupts enablein SPIRIT1, commands required to ready the hardware and payload. For that we have to take the help from SPIRIT1 datasheet.

4.4.9.0.2   Interrupts in SPIRIT1   A handler is expected to perform any necessary acknowledgement of the parent IRQ via the correct chip specific function. In the Linux there is already the entry of Interrupts under this directory
#include<irq.h>
.

- set_irq_chip(irq, chip)
  Set the mask/unmask methods for handling this IRQ.

- set_irq_handler(irq, handler)
  Set the handler for this IRQ (level, edge, simple).

- set_irq_chained_handler(irq, handler)
  Set a "chained" handler for this IRQ - automatically enables this IRQ.

- set_irq_flags(irq, flags)
  Set the valid/probe/noautoenable flags.

- set_irq_type(irq, type) Set active the IRQ edge(s)/level. This is also for the GPIO of the SPIRIT1. As we seen in the datasheet of the SPIRIT1 that there are four GPIO's in it. This interrupt is for the GPIO's. set_GPIO_IRQ_edge() is obsolete, and should be replaced by set_irq_type.

4.5   Testing

The driver job is to send the commands from the one device to another one and getting back its response. If all this happen sucessfully then that should means that the one SPIRIT1 is able to

communicate with another SPIRIT1.

For testing the SPIRIT1 we have to do the following procedure:

4.5.1   Enable the SPI in RPI2 board

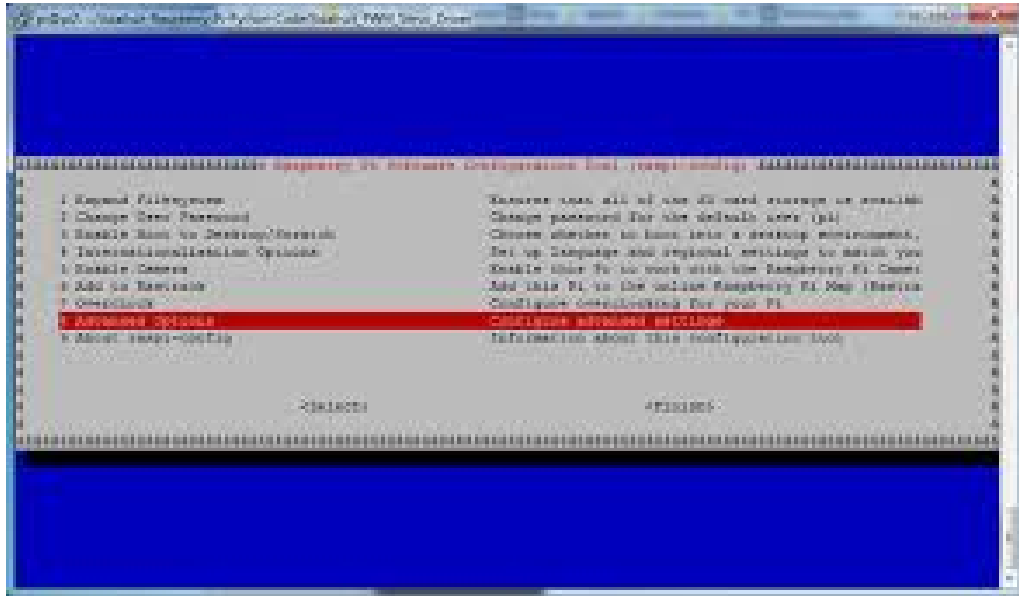1. To enable the SPI in raspberry PI board. Run the following command. **sudo raspi-config**

Figure 4.8: enable the SPI in RPI2 board

2. Select the Advanced boot option

3. Select Enable the SPI and then select on save and exit

4.5.1.1   Make the entry in dts

We discused in the previous section about the .dts. Now we have to make the entry of the SPIRIT1 in the dts then we have to convert this into .dtb. for making the entry in the .dts we have to run the following command:

cd linux

cd arch/arm/boot/dts

vi bcm2709-rpi2b.dts

then add the following in the SPI category as shown in figure:

```
spidev1: spidev@1{
    compatible = "stsp";
    reg = <1>;
#address-cells = <1>;
#size-cells = <0>;
    spi-max-frequency = <500000>;
  };
};
```

Figure 4.9: Entry in dts

4.5.2   Interfacing between RPI2 and SPSGRF(SPIRIT1)

The SPSGRF-868 and SPSGRF-915 are easy-to-use, low power sub-GHz modules based on the
SPIRIT1 RF transceiver, operating respectively in the 868 MHz SRD and 915 MHz ISM bands.
The SPSGRF series enables wireless connectivity in electronic devices, requiring no RF experience
or expertise for integration into the final product. As an FCC, IC, and CE certified solution,
the SPSGRF series optimizes the time-to-market of end applications. The SPSGRF-915 is an
FCC certified module (FCC ID: S9NSPSGRF) and IC certified (IC 8976C-SPSGRF), while the
SPSGRF-868 is certified CE0051.

Pin name	RPI2 pin no.	SPSGRF pin no.
MISO	19	12
MOSI	21	12
SCLK	23	12
CSn	24	12
VSS	1	12
GND	6	12

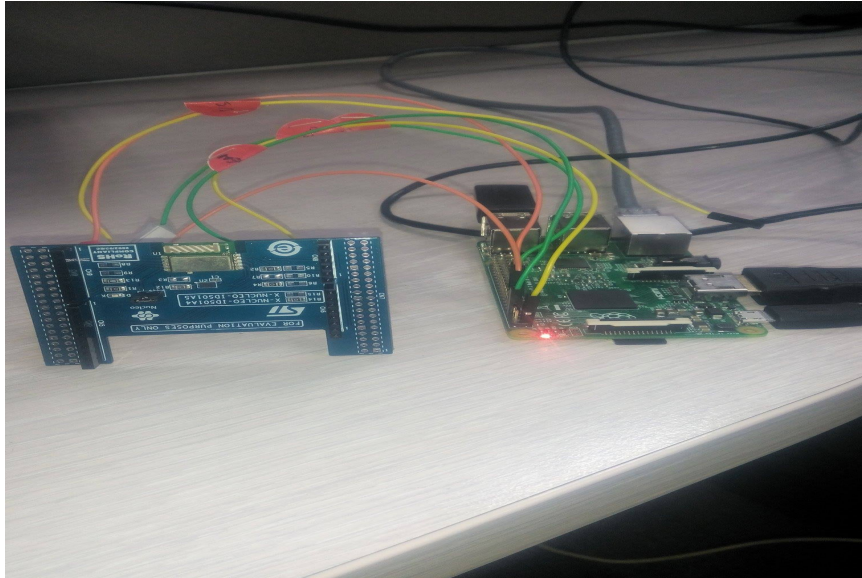Table 4.1: RPI2 and SPIRIT1 Interfacing

Figure 4.10: Interfacing between RPI2 and SPIRIT1

4.5.3 Test using logic Analyzer

We discussed in the previous section about the logic analyzer and how it works. Now we test the SPI of SPIRIT1 using logic analyzer. It gives us an idea that the data through the SPI is going correctly or not. For that we have to connect the logic Analyzer with the SPIRIT1 board and send the data over sysfs file using command:

echo FF > filename

Run the above command and notice the data over logic analyzer. It gives: After the data through the SPI is flowing correctly. Now we have to send the commands for the SPIRIT1 to become ready. For that we send the command 0x03 over SPI after that SPIRIT1 become ready to accept the data.

Figure 4.11: Raspberry PI output on logic Analyzer
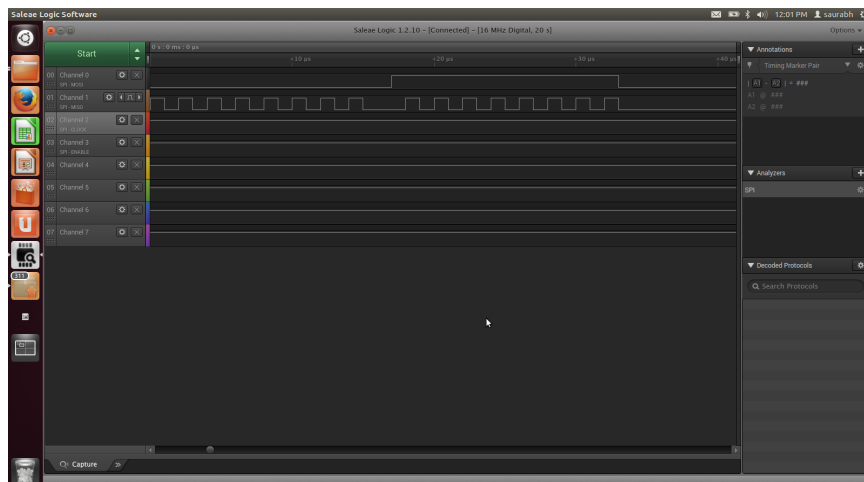
4.5.4 Test the hardware

When the SPIRIT1 become ready to accept the data. Now for the hardware verification we have to read the version info and version number of the SPIRIT1 to verify that whether our hardware is recognised by the Linux or not.

As shown in the table that for the version info and version number we have to read the value

Regsiter	Address	Field name	Reset	Description
DEVICEINFO	0xF0	PARTNUM	0x01	Device part number
DEVICEINFO	0xF1	VERSION	0x30	Device version number

Table 4.2: Hardware Register

from the SPIRIT1 using the read function and that should give the value,

Part NUMBER	Part VERSION
1	48

Table 4.3: Value to be read from SPIRIT1

5.1 LED7708 Introduction

The LED7708 has been specifically designed to supply several LEDs from a single low-voltage rail in order to address TV and monitor backlights, medium- and large-size LCD panel backlights and RGB/RGGB backlight applications. It has sixteen current generators rated at 40 V and a 4-wire serial interface to control LED brightness. These channels can be put in parallel for higher output currents. A selectable 12-bit or 16-bit gray-scale brightness control allows independent PWM on each channel. A programmable on-chip dimming oscillator is provided to simplify external circuitry. The device has dedicated pins to lock synchronize with other devices (master or slave) for noise reduction in multi- device applications.

The LED7708 implements basic protection (OVP, OCP and thermal shutdown), as well as LED-array protection. It can detect and manage open-LED and shorted-LED faults, and different fault management options are available in order to cover most application needs. The board has been designed as a demonstration of a solution for medium/large LCD panel backlight drivers, but is suitable for any application involving several LEDs assembled in strings (e.g. advertisement panels, signs, gaming, etc.).

5.1.1 LED7708 read/write protocols

Protocols for read and write LED7708 registers.

5.1.1.1 Serial interface internal registers

The internal registers are organized in control registers and brightness registers. Access to the desired register is performed by encoding the destination address via a serial key on the bus (a serial key consists of a certain number of clock pulses during the high-state of the LE signal). All the internal registers are 16 bits wide.

DCLK rsing edge with LE=1	operation
1	Data latch
2-3	Global latch
4-5	Write CHSEL
6-7	Read CHSTA
8-9	Write DEVCFG
10-11	Read DEVCFG
11-12	Write GSLAT

Table 5.1: Internal register operation encoding

5.1.2 LED7708 Linux device driver

Below is the Linux kernel Architecture showing where all the basic components and their functionality lies and showing where the LED7708 LDD fits in.
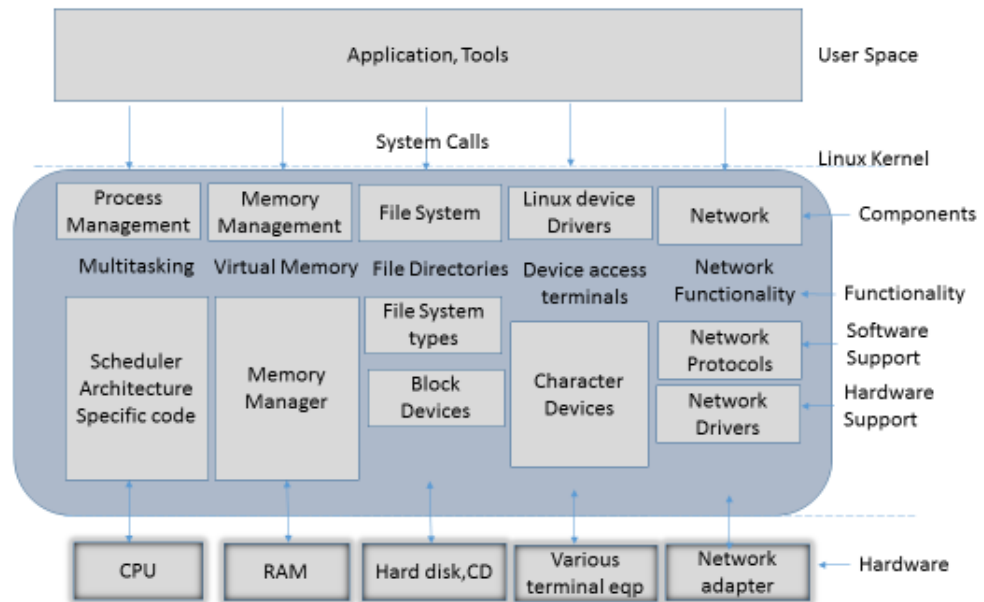

Figure 5.1: Linux kernel architecture

5.2   Interfacing of the LED7708 with RPI2 board

5.2.1   Hardware setup

You will need the following items while working with LED7708 with Raspberry Pi 2

1. **LED7708** Evaluation Board.

2. A Connector or Jumper wires to connect LED7708 with the Rpi2 according to the connection diagram shown in figure below.

3. Standard HDMI cable for display along with a HDMI Monitor or also can alternatively work remotely on PC (Windows or Linux) using any remote terminal Application like Putty.

4. RS232 FTDI hardware cable for taking serial Logs (Only for taking boot time debugging logs).

5. Wireless/Wired Keyboard for running the test scripts if working directly (Not remotely) on the RPI2 board.

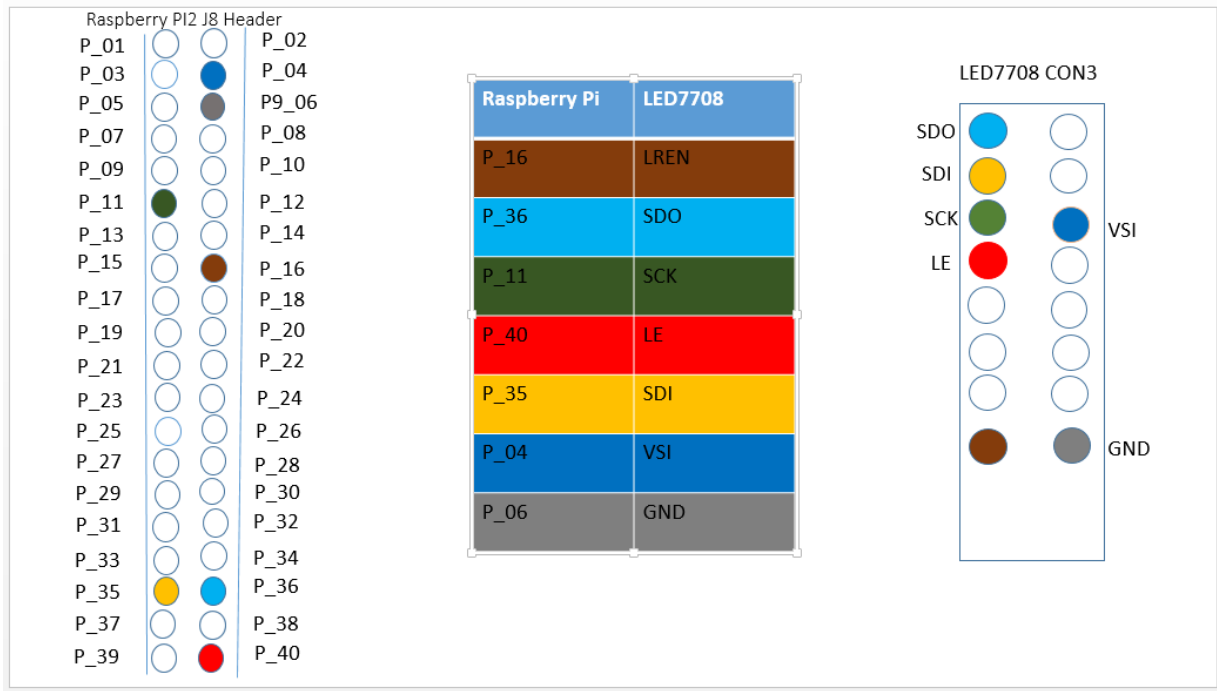### 5.2.1.1   Hardware connection between LED7708 and RPI2 board

Figure 5.2: Interfacing between LED board and RPI2

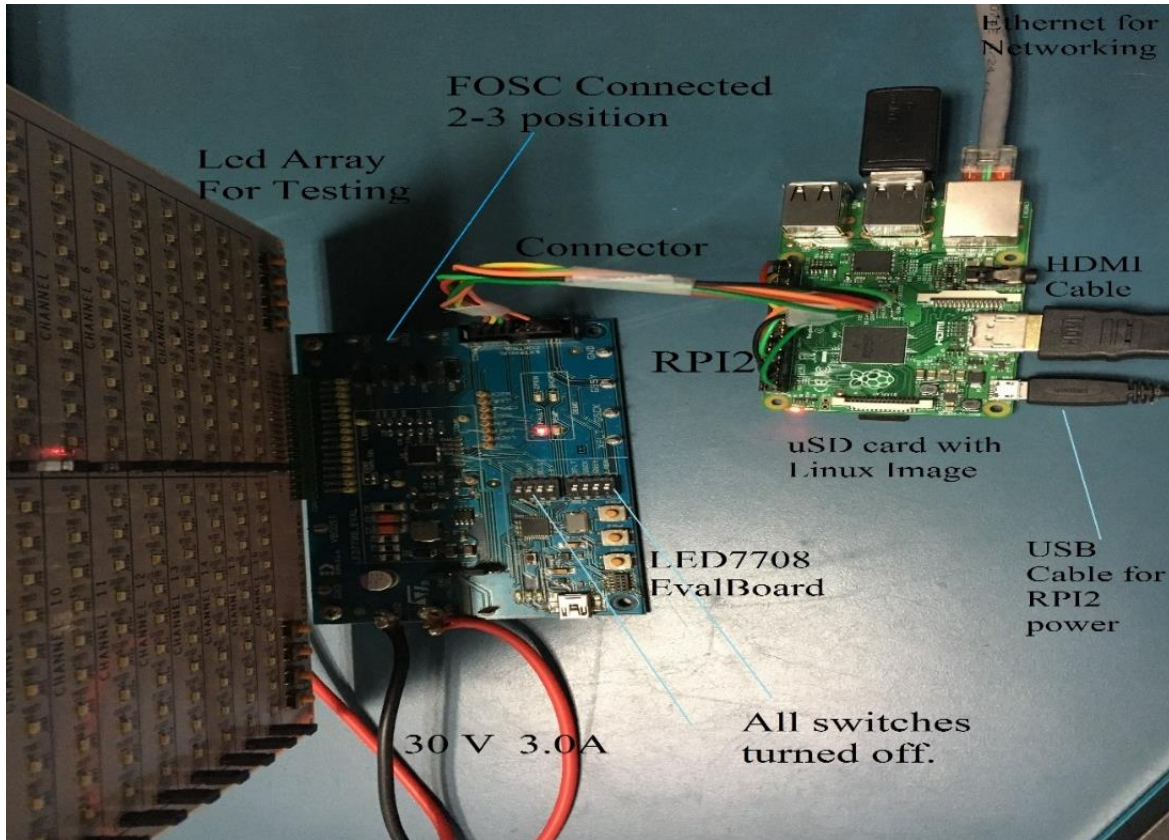### 5.2.1.2   Hardware Setup between RPI2 and LED7708 Eval board

Figure 5.3: Hardware setup between LED7708 and RPI2 board

5.2.2   Software setup

For integrating the kernel driver with the Raspberry pi we have to bring up the RPI2 with the appropriate Linux kernel. Please follow the below steps to prepare the RPI2 with Kernel 4.4.23 version.

1. Download the kernel source Create a directory to store the Raspberry related files (e.g. /home/<my pc>/rpi2) and run the following command in that directory.
   git clone https://github.com/raspberrypi/linux.git
   if you want to branch to any other kernel run the below command .e.g below command will clone 4.8.y branch. To see the available branches you can do git branch -a
   git checkout -b rpi-4.8.y –track origin/rpi-4.8.y

2. Download the tool chain Tool chain will be required to build/compile the boot loader and the kernel sources. Download Raspberry PI cross-compilers by running the following command on your build machine.

git clone https://github.com/raspberrypi/tools

3. Configuring the kernel
   cd linux
   KERNEL=kernel7
   make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2709defconfig

4. Building the kernel Now it's time to build the kernel (image) and modules and .dtbs. Run this command in your terminal to build the kernel.
   make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs

5. Flashing the kernel on a new SD card Before that if you are using the new SD card you have to flash the NOOB or Raspbian image on it first and then follow the below steps.

   - Download Raspbian image directly. using a computer with an SD card reader, visit the official Raspberry Pi Downloads page.

   - Click on the Rasbian.

   - Click on the Download ZIP button under Raspbian Jessie (full desktop image), and select a folder to save it to.

   - Extract the files from the zip.

   - Visit **etcher.io** and download and install the Etcher SD card image utility.

   - Run Etcher and select the Raspbian image you unzipped on your computer or laptop.

   - Select the SD card drive. Note that the software may have already selected the right drive.

   - Finally, click Burn to transfer Raspbian to the SD card. You'll see a progress bar that tells you how much is left to do. Once complete, the utility will automatically eject/unmount the SD card so it's safe to remove it from the computer.

5.3   Adding Linux Device driver support in the Linux kernel

Adding a LDD support in the Linux kernel means adding its source code in the Linux kernel. This can be done either dynamically (building the driver as a module out of the tree kernel building) or by compiling the driver along with the kernel sources (in built driver). Also before that Hardware (LED7708) specific parameters (like interface signals with the Raspberry PI) needs to be defined to the kernel in the device tree of the platform in which LED7708 needs to be interfaced.

5.3.1   Adding Device tree bindings for LED7708 device driver

Adding Device tree bindings for the Led7708 Linux driver.

In <path_to_kernel>/arch/arm/boot/dts/

nano bcm2709-rpi-2-b.dts

In this file make the entry for the led7708 board in the led class.

5.3.2   Adding Linux driver in the kernel - Out of the tree

Use the below Makefile to build the module.

#MODULES := leds-st.o

ARCH := arm

CROSS_COMPILE := arm-linux-gnueabihf-

obj-m := $(MODULES)

OUTDIR := /<path to built kernel>/linux/

MAKEARCH := $(MAKE) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE)

all: modules

modules:

$(MAKEARCH) -C $(OUTDIR) M=$shell pwd modules

clean:

$(MAKEARCH) -C $(OUTDIR) M=$shell pwd clean

5.4   Insert the modules

Modules can be inserted using the commands

insmod leds-st.ko

Once the module is inserted successfully you can see the sysfs being created in the led directory and there is the sixteen channels for the leds class which you can access directly and build your application by accessing them.

Example wave forms for the below sequence of commads:

- W(08,04,2130)

- W(08,04,8440)

- W(04,04,FFFF)

- W(08,04,2131)

- W(02,04,FFFF)
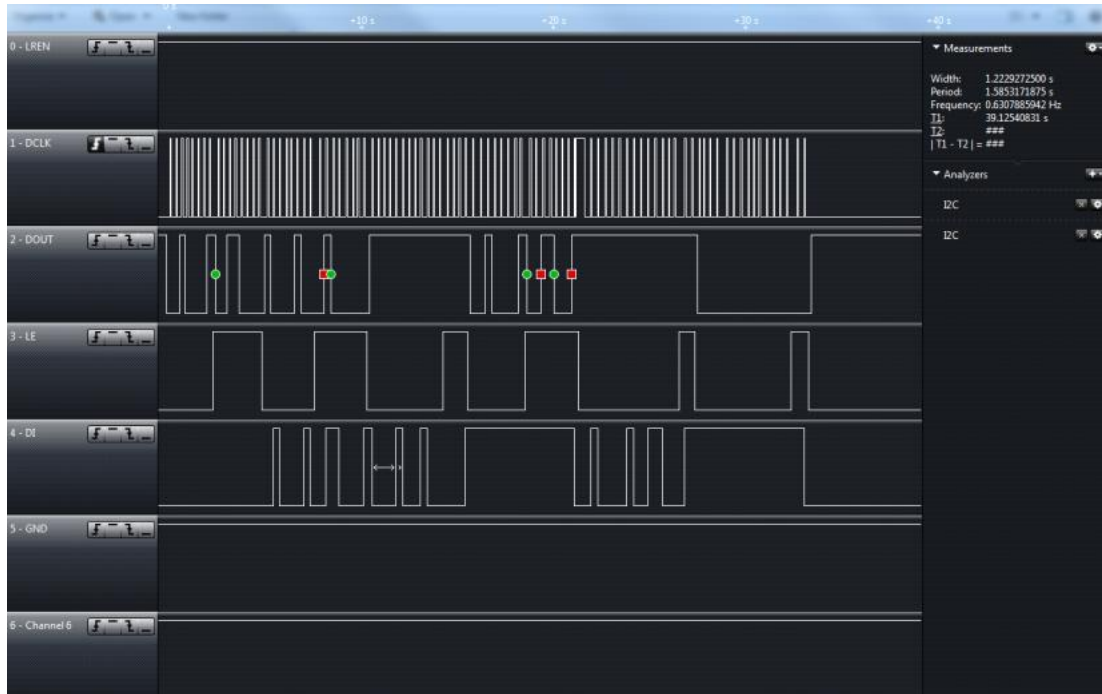
Running the following scripts give the below waveform.

Figure 5.4: Wave form for the example script

CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

I have learnt a great deal working on this project. The learning was not limited to project only but the whole experience of working as an intern in a multinational company like STMicroelectronics was immensely educational. Being an intern, one is always challenged by the fundamental difference between classroom coaching and real industrial experience. But such a challenge is exactly the purpose of six months training.

The whole experience of working on this project and contributing in a few others has been very rewarding as it has given great opportunities to learn new things and get a firmer grasp on already known technologies. Here is a reiteration of some of the technologies I have encountered, browsed and learnt:

- Operating System: **Linux**

- Language: **C**

- Development Host: **Raspberry Pi**, **BeagleBone Black**

- Integrated Build Environment: **Buildroot**

So during this project I learnt all the above things. Above all I got to know how software is developed and how much work and attention to details is required in building even the most basic of components of any project. Planning, designing, developing code, working in a team, testing, etc. these are all very precious lessons in themselves.

6.2 Future Scope

SPIRIT1 is a relatively new technology. This implies that there is a long road ahead of more features, more research and more hardware for this technology. As such there is always room for inclusion in the Linux kernel for all of this. If the completely new hardware is introduced then it becomes very important to have its driver included in Linux. This is not just have the support for the device in Linux but it also acts like an endorsement for the device that its driver has been registered with the coveted Linux kernel.

Talking specifically about some of the other solutions by ST, following two come to immediate consideration that can also be driven by a Linux host by means of device driver written in a manner much similar to SPIRIT:

1. **SPIRIT 1**

2. **S2-LP**

The continuous need for more software related to SPIRIT1 in Linux is not just restricted to device driver development. WPAN is the official stack for Linux but even it does not implement everything in the WPAN we have to include the features of the IEEE 802.15.4. Some features are missing and are intended for the future. Also since with new specification releases, more and more features get introduced into the technology. So the stack will also have to sustain with the specification. Some of the things that are not yet implemented in WPAN but are part of the IEEE 802.15.4:

- New netlink framework nl802154

- Privacy

- IEEE802154 cryptography layer on top of nl802154

- Improvements in frame parsing and creation

- Better connection management between linux and other OS such as Contiki

[1] **IEEE802.15.4 specification**
http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf

[2] **Linux Device Drivers**, 4th edition
a book by *Jonathan Corbet, Alessandro Rubini*, and *Greg Kroah-Hartman*.

[3] **Linux Kernel Documentation for device drivers**.

[4] **SPIRIT1 datasheet**
http://www.st.com/content/ccc/resource/technical/document/datasheet/68/6c/7b/ec/b2/6b/49/16/DM

[5] **Build the kernel for BBB**
https://eewiki.net/display/linuxonarm/BeagleBone+Black