

A USB-BASED REAL-TIME COMMUNICATION INFRASTRUCTURE FOR ROBOTIC PLATFORMS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Cihan Öztürk

August, 2009

I certify that I have read this thesis and that in my opinion it is fully adequate,
in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Uluç Saranlı (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate,
in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Afşar Saranlı

I certify that I have read this thesis and that in my opinion it is fully adequate,
in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Selim Aksoy

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

A USB-BASED REAL-TIME COMMUNICATION INFRASTRUCTURE FOR ROBOTIC PLATFORMS

Cihan Öztürk

M.S. in Computer Engineering

Supervisor: Assist. Prof. Dr. Uluç Saranlı

August, 2009

A typical robot operates by carrying out a sequence of tasks, usually consisting of acquisition of sensory data, interpretation of sensory inputs for making decisions, and application of commands on appropriate actuators. Since this cycle involves transmission of data among electro-mechanical components of the robot, high quality communication is a fundamental requirement. Besides being reliable, robust, extensible, and efficient, a high quality communication infrastructure should satisfy all additional communication requirements that are specific to the robot it is used within. To give an example, for a rapid moving autonomous robot with a reactive controller which is intended to be used in time critical situations, a real-time communication infrastructure which guarantees demanded response times is required.

The Universal Robot Bus (URB) is a distributed communication framework based on the widely used I^2C standard, intended to be used specifically within rapid autonomous robots. Real-time operation guarantees are provided by defining upper bounds in response times. URB facilitates exchange of information between a central controller and distributed sensory and actuator units. Adoption of a centralized topology by connecting distributed units directly to a central controller creates a bottleneck around the central unit, causing problems in scalability, noise and cabling. In order to overcome this problem, URB is physically realized such that gateways (bridges) are incorporated between the central and distributed units which offload the work of the central unit and master the underlying I^2C bus. Connection between the central unit and the gateway, the uplink channel, can be established using any high bandwidth communication alternative which successfully satisfies communication requirements of the system.

The main contribution of this thesis is the design and implementation of the

URB uplink channel using the well known Universal Serial Bus (USB) protocol. Although true real-time operation is not feasible with USB due to its polling mechanism, USB frame scheduling of 1ms is acceptable for our application domain. In this thesis, hardware components used in the USB uplink implementation as well as our software implementation are covered in detail. These details include the firmware running on the gateway, a Linux based device driver and a client control software that uses a USB library running on central controller, and finally sub-protocols between the application-driver and driver-firmware layers. The thesis also includes our experiments to estimate the performance of the USB uplink in terms of its roundtrip latency, bandwidth, scalability, robustness, and reliability. Finally, this thesis also serves as a reference on distributed systems, device driver development, Linux kernel programming, communication protocols, USB and its usage in real-time applications.

Keywords: USB, real-time communication, distributed systems, URB.

ÖZET

ROBOTİK PLATFORMLAR İÇİN USB TABANLI GERÇEK ZAMANLI BİR İLETİŞİM ALTYAPISI

Cihan Öztürk

Bilgisayar Mühendisliği, Yüksek Lisans

Tez Yöneticisi: Yard. Doç. Dr. Uluç Saranlı

Ağustos, 2009

Robotlar genel olarak algılayıcılardan veri toplama, algısal girdileri karar almak için değerlendirme, ve komutları uygun eyleyiciler üzerinde uygulama şeklinde bir dizi görevleri yerine getirerek çalışmaktadırlar. Bu döngü verilerin robotun elektro-mekanik bileşenleri arasındaki iletimini gerektirdiğinden, yüksek kalitede bir iletişimi sağlama temel bir gereksinimdir. Kaliteli bir iletişim altyapısı güvenilirlik, dayanıklılık, geliştirilebilirlik ve verimlilik gibi özelliklerin yanı sıra, kullanıldığı robotun bütün iletişim ihtiyaçlarını karşılayabiliyor olmalıdır. Örneğin, hızlı hareket eden, tepkin bir kontrol birimine sahip olan ve kritik zamanlı işlerde kullanılan özerk bir robotun istenen tepki sürelerini garanti eden gerçek zamanlı bir iletişim altyapısına sahip olması gereklidir.

Evrensel Robot Veriyolu (URB), dağıtık kontrol sistemleri için tasarlanmış I^2C tabanlı bir iletişim çatısıdır ve özellikle hızlı hareket eden özerk robotlarda kullanılmak üzere geliştirilmiştir. Kullanıcılara gerçek zamanlı işlem olanağı tepki zamanlarında üst limitler tanımlanarak verilmiştir. URB, merkezi bir kontrol birimi ile dağıtık bulunan algılayıcı ve eyleyici birimleri arasındaki veri iletimine olanak tanır. Merkezi bir topolojiyi benimseyerek dağıtık uç birimleri doğrudan merkezi kontrol birimine bağlamak, merkezi birim etrafında bir darboğaz yaratmakta, özellikle ölçeklenebilirlik, gürültü ve kablolama hususlarında sorunlar yaratmaktadır. Bu problemi aşmak için, URB'nin fiziksel gerçekleştiriminde merkezi ve uç birimler arasına bir ağ geçidi (köprü) yerleştirilmiştir. Bu ağ geçidi merkezi birimin iş yükünü hafifletmekte ve bağlı bulunduğu I^2C veriyolunu yönetmektedir. Merkezi birim ve ağ geçidi arasındaki üst bağlantı kanalı olarak adlandırdığımız bağlantı, iletişim gereksinimlerini karşılayan yüksek bant genişliğine sahip herhangi bir iletişim seçeneği ile gerçekleştirilebilir.

Bu tezin temel katkısı URB üst bağlantı kanalının Evrensel Seri Veriyolu (USB) kullanarak tasarım ve gerçekleştirmedir. USB'nin yoklama düzeniğinden dolayı gerçek zamanlı işlem tam anlamıyla mümkün olmasa bile, 1ms'lik yoklama peryodu uygulama alanımız için kabul edilebilir bir değerdir. Bu tezde, URB üst bağlantısının fiziksel gerçekleştirmede kullanılan bazı donanım bileşenleri ile geliştirilen yazılımların detaylı açıklamaları yer almaktadır. Bu detayların başlıcaları ağ geçidinde çalışan bellenim, merkezi birimde çalışan Linux tabanlı aygıt sürücü ve USB kütüphanesini kullanan bir istemci kontrol yazılımı, ve son olarak uygulama-sürücü ve sürücü-bellenim arasında yer alan alt-protokollerdir. Bu tez ayrıca USB üst bağlantısının başarımını gidiş-geliş gecikmesi, bant genişliği, ölçeklenebilirlik, dayanıklılık ve güvenilirlik gibi ölçütlere göre belirlemeye çalışan deneylerin sonuçlarını da içermektedir. Son olarak, bu tez aygıt sürücü geliştirimi, Linux çekirdek programlama, iletişim protokolleri, USB ve gerçek zamanlı uygulama alanları gibi konularda kaynak niteliğindedir.

Anahtar sözcükler: USB, gerçek zamanlı iletişim, dağıtık sistemler, URB.

Acknowledgement

First of all, I would like to thank my supervisor Assist. Prof. Dr. Uluç Saranlı for his guidance and support throughout my study. I am grateful to his tremendous patience and endless enthusiasm in conveying me his academic knowledge and experiences. He has been more than generous for sacrificing his time to teach me how to approach problems scientifically, realize high quality work, and deliver proper presentations. During my research, I felt the confidence of working with him all the time, who always had some brilliant ideas for the problems I faced, and was so tolerant that he never scorched me even if I had not put enough effort in doing my responsibilities. I feel lucky to be one of his first graduate students, possibly benefiting from his teaching skills one of the most, and thank him one more time for the opportunities he has provided me.

Second, I am very grateful to all members of SensoRhex Project, specifically to Assist. Prof. Dr. Afşar Saranlı, Assist. Prof. Dr. Yiğit Yazıcıoğlu and Professor Dr. Kemal Leblebicioğlu from Middle East Technical University for their help and insights. I also would like to thank Haldun Komşuoğlu for sharing his studies and ideas with us generously.

I am also thankful to Akın Avcı, for being a perfect colleague, helping me any time I ask, and leaving behind many academic work on which I base my thesis, Özlem Gür for being a good friend always ready for help, Ömür Arslan being so easy-going and for his endless efforts in organizing some group activity, Sıtar Kortik for being a buddy who participates in any social activity whenever I call him, Tuğba Yıldız for being a cheerful chatter with whom I never get bored, İsmail Uyanık for feeding me with his junk food during my late at night studies in the lab, Mert Duatepe for advising and helping me to come to Bilkent and work with my supervisor, Bilal Turan, Tolga Özaslan, Utku Çulha and all other members of Bilkent Dexterous Robotics and Locomotion Group.

I also would like to thank all members of linux-usb mailing list, specifically to Alan Stern, for sharing their ideas with me, which helped me very much in solving problems related with this thesis.

I am grateful to my current bosses Seçkin Tunahılar and Derya Arda Özdamar for showing tolerance during the period of writing this thesis.

I am also appreciative of the financial support from Bilkent University, Department of Computer Engineering, and TÜBİTAK, the Scientific and Technical Research Council of Turkey.

Graduate study at Bilkent Campus was a such a wonderful experience that I am sure I will be missing those days for the rest of my life. I thank to all of my friends who shared these good memories with me. I cannot mention all the names since there are so many, but I specifically thank to Burak, for being a perfect home-mate for almost three years.

Last, but not least, I would like to thank my parents, brother and sister-in-law for the opportunities they had provided to me, as well as for their endless love, support and encouragement.

Contents

1	Introduction	1
1.1	Robotic Systems	1
1.2	Digital Control Systems	2
1.3	Related Communication Protocols	3
1.4	Motivation and Contributions	6
1.5	Outline	8
2	Background	9
2.1	I/O with Peripheral Devices	10
2.2	Device Drivers	11
2.3	Linux Kernel Programming	14
2.3.1	Character Drivers	16
2.3.2	Concurrency Management	20
2.3.3	Synchronous I/O	23
2.4	The Universal Serial Bus	24

2.4.1	Bus Topology	25
2.4.2	Communication Flow	26
2.4.3	Data Transfer Types	29
2.4.4	Packet Transmission	32
2.4.5	USB Descriptors	32
2.5	USB Device Drivers for Linux	33
2.6	USB module of SiLabs F340 Board	37
3	The Universal Robot Bus	39
3.1	URB Overview	40
3.2	URB Components	41
3.2.1	URB Nodes	42
3.2.2	URB Bridge	43
3.2.3	The URB CPU	45
3.3	URB Communication Model	45
3.3.1	Uplink Communications	45
3.3.2	Downlink Communication	50
3.4	URB APIs	51
3.4.1	URB CPU API	51
3.4.2	URB Node API	53
4	USB Uplink for URB	54

4.1	Communication Decisions	56
4.1.1	Application-to-Driver Subprotocol	56
4.1.2	Driver-to-Firmware Subprotocol	57
4.1.3	The Uplink Transfer Policy	58
4.2	Bridge Firmware	60
4.3	The Linux USB Device Driver for URB	62
4.3.1	Driver Functionality	64
4.3.2	Buffering Decisions	67
4.3.3	Concurrency Management Issues	69
4.3.4	Algorithm	70
4.4	URB Control Software	74
4.4.1	USB Library	74
4.4.2	User Application	75
5	Performance Analysis of the USB Uplink	77
5.1	Experimental Background	77
5.1.1	Analysis Model	77
5.1.2	Analysis Goals	80
5.1.3	Test Software	82
5.1.4	Mathematical Background	83
5.2	Experimental Setup	86

5.2.1	Configuration of the URB CPU	86
5.2.2	Configuration of the URB Bridge	87
5.3	Estimation of Host USB Characteristics	87
5.4	Experiments and Results	88
5.4.1	Single Bridge Tests	88
5.4.2	Multiple Bridge Tests	94
6	Conclusion	98
A	URB Downlink Details	103
A.1	Downlink Control with Inbox/Outbox 0	103
B	URB Uplink Details	105
B.1	URB Uplink Bridge Commands	105
B.2	URB Uplink Packet Formats	109
B.3	An Example User Application (CPU Side)	109
B.4	Core Algorithm	114

List of Figures

1.1	Arrangement of eight nodes in central and distributed architectures.	6
2.1	Typical layered decomposition of an I/O system.	12
2.2	Physical Bus Topology of USB [8].	26
2.3	Logical Topology of USB [8].	27
2.4	Interlayer Communications Model for the USB standard [8]. . . .	28
2.5	Internal Layout of a USB Host [8].	29
2.6	USB Descriptor Hierarchy [11].	33
2.7	General layout of a USB Host Stack [2].	35
2.8	USB FIFO allocation for the C8051F340 microcontroller [27]. . . .	38
3.1	Logical Topology of a URB system as seen by the user [4].	40
3.2	Physical Topology of a URB system [4].	41
3.3	Internal software structure of a URB Node [4].	42
3.4	The layout of the of the URB Bridge Firmware.	43
3.5	URB Transaction types.	47

3.6	Packet types for the URB Uplink.	47
3.7	The URB Generic Uplink Packet Layout.	49
3.8	The CPU API.	52
4.1	A layered component decomposition of the USB uplink for the URB system.	54
4.2	Simplified software layout for the USB Uplink of URB.	55
4.3	General request packet format for the Application-to-Driver Sub-protocol.	56
4.4	The processing of a packet by the driver: incoming packet is interpreted by the driver according to the operation header, operation header is discarded, and remaining payload is buffered.	57
4.5	State Machine of the write function.	65
4.6	Timeline of events initiated by a call to the write function within the write thread.	71
4.7	Activity diagram showing the flow of events involved in the write function.	72
4.8	Activity diagram showing the flow of events involved in callback functions of the write thread.	73
4.9	Activity diagram showing the flow of events involved in the read function.	74
5.1	Structure of a frame with bulk transactions as a data block. . . .	78
5.2	Change of number of unprocessed requests over number of nodes based on usbmon output for 1000 request submissions. Number of unprocessed requests is 0 for n=1:6.	90

5.3	Comparison of time versus transfer size for 3 transfers with n:1,6,14, based on usbmon output.	91
5.4	Mean and standard deviation values of round-trip latencies for 1000 request submissions to each node, obtained from usbmon. Note that y-xis of the plot is scaled between 1600-2000ms in order to display the standard deviations more clearly.	92
5.5	Round-trip latencies of 1000 requests for n=6, obtained from usbmon.	94
5.6	Number of unprocessed requests over number of nodes for (a) bridge 1. (b) bridge 2. Number of unprocessed requests is 0 for n=1:6 for both bridges.	95
5.7	Mean and standard deviation values of round-trip latencies for 1000 request submissions to each node of (a) bridge 1 (b) bridge 2, obtained from usbmon. Note that y-xis of the plot is scaled between 1300-2100ms in order to display the standard deviations more clearly.	96
B.1	URB Uplink Request and Response Packet Formats.	109
B.2	Activity diagram showing flow of events involved in all threads. Interaction between the threads, signaling blocked threads, are in- dicated by arrows across the lanes.	115

List of Tables

2.1	USB Transfer Types	31
4.1	Endpoint Configurations of the Bridge Firmware. Direction is from the perspective of the CPU.	61
5.1	Test Environment Settings	86
5.2	Comparison of data from usbmon and application.	89
5.3	Mean and standard deviation values of round-trip latencies for 1000 request submissions to each node, obtained from usbmon. . .	93
5.4	Various transfer statistics of two bridges for n=6 and 1000 request submissions, obtained from usbmon.	97
5.5	Various transfer statistics of three bridges for n=6 and 1000 request submissions, obtained from usbmon. Bridges 1 and 2 are connected to one root hub, and Bridge 3 is connected to another root hub. .	97
B.1	Bridge command types with description	105

Chapter 1

Introduction

This thesis proposes a USB based high bandwidth real-time communication connectivity for a small-scale distributed control system. The distributed control system that the thesis is based upon is called the Universal Robot Bus (URB) [4, 5], intended to be used specifically in mobile autonomous robot applications.

1.1 Robotic Systems

A robot is an electronically re-programmable, multi-tasking electro-mechanical system, often capable of carrying out a wide range of motions or tasks, typically, but not exclusively by autonomous means [26]. The main difference of robots from general embedded systems is their programmable nature and their capability for environmental sensing.

Robots can be classified based on different criteria, such as their level of autonomy, type of motion, path control and so on. According to the classification by level of autonomy, robots are categorized into semi-autonomous and autonomous robots. While a semi-autonomous robot is usually teleoperated or remotely controlled by a human operator, a fully autonomous robot has autonomy in all aspects such as energy, computation, sensing and actuation so that

it can perform desired tasks in unstructured environments without continuous human guidance.

Robots can be controlled by four common control paradigms which are reactive, deliberative, hybrid and behavioral. While a reactive robot has local sensors to obtain the stimulus, and respond with a reflexive reaction to the input, a deliberative robot requires a global planning based on an internal model of the environment [28]. Since reactive control allows reacting very quickly to state changes, it is appropriate for autonomous robots that move rapidly in outdoor environment. Other control paradigms include a combination of the two aforementioned methods, hybrid being the usage of both reactive and deliberative paradigms together, and behavioral being the usage of one of the two based on current state.

Among the diverse classes of robots with different control paradigms, we are specifically interested in reactive autonomous robots such as RHex [24]. Since RHex is intended to be used in missions that are hazardous for human, such as search and rescue, humanitarian demining, military and planetary exploration, it requires a communication infrastructure with capabilities of real-time communication, modular and simple hardware and software design to support extensibility, reliability and robustness.

1.2 Digital Control Systems

Most, if not all, robotic and embedded systems incorporate a digital control mechanism. Digital control mechanisms are mostly implemented using two major architectures, central and distributed. While all components in the former, which might be microcontrollers driving sensors or motors, are directly connected to the master, the latter architecture has components distributed along a communication bus or network. It is relatively easier to design and implement software for a central control system, since a single master controls all I/O operations. In contrast, a distributed controller needs to gather information from all nodes and

send commands to them while slave nodes are independently processing I/O operations as well as communicating with the master.

Nonetheless, various drawbacks of the central architecture make distributed systems preferable. One disadvantage is that cabling around the master controller causes physical implementation problems, and it practically becomes impossible to implement a central system with many connected nodes due to space limitations. Another problem is increased noise around the master due to extensive wiring, which degrades the reliability of the entire system. Moreover, the central architecture begins performing poorly as the number of connected nodes increases, and is hence not scalable. On the other hand, although a distributed architecture has its own problems such as collisions on the shared bus and software complexity, it generally provides a much more scalable and extensible solution. In a distributed system, new nodes can easily be added to any available slot along the shared bus, eliminating problems with physical connections.

1.3 Related Communication Protocols

Distributed systems need to maintain reliable flow of information across distributed components in order to reliably preserve the functionality of the entire system. This critical flow of information is established through the means of communication standards, realized using different physical connectivity alternatives. A distributed system often implements its internal communication by introducing a shared bus that connects its components. In this section, we discuss various communication protocols that can potentially be used in the URB architecture, evaluating their compliance in terms of providing real-time, robust and reliable communication at high bandwidth.

The *Controller Area Network (CAN)* [21] is an asynchronous serial communication bus standard designed for networking in industrial systems in need of data rates up to 1Mbps, and high levels of data integrity. Because of its strong immunity to noise, CAN is widely used in robots where power lines generate too much

noise [19]. Although CAN is one of the mostly used protocols for distributed real-time applications, its complex software development process leads us to evaluate other protocols.

The *Inter-Integrated Circuit (I^2C)* [20] is a serial communication bus used to attach low-speed peripherals to a microcontroller, allowing data rates from 10Kbps to 3.4Mbps. Since the I^2C interface is built into many microprocessors, it is easy to downsize nodes. For this reason, it is widely used in small robots that have strong size constraints [17].

The *Serial Peripheral Interface (SPI)* is a synchronous serial data link standard that operates in full duplex mode. SPI tends to be more efficient than I^2C in point-to-point applications due to having less overhead, but it requires more pins on integrated circuit packages than I^2C , which brings up cabling problems.

The *Recommended Standard 232 (RS232)* [9] is an asynchronous serial communication protocol which has been widely used for serial communications until new standards such as USB and Firewire started to emerge. It is still used to connect older designs of peripherals allowing bit rates between 1200-230400 baud, which is too low for our requirements.

The *Industry Standard Architecture (ISA)* is a synchronous parallel bus standard with a theoretical transmission capacity of 8 MB/s. Although its throughput is suitable for our needs, the complexity of the standard in terms of both physical connectivity and software makes it inappropriate for our application domain.

The *Peripheral Component Interconnect (PCI)* is a synchronous bus developed as a replacement for the outdated ISA bus, allowing a peak transfer rate of 2133 MB/s with its new implementations. Since PCI is a parallel bus standard as the ISA bus, it is not suitable for our work either.

The *Universal Serial Bus (USB)*, described in detail in Section 2.4, is a bus standard developed with the intention of providing a flexible and low cost protocol with high transmission rates.

Ethernet is a global network standard used for local area and wide area network connections, having speeds of up to 1Gbps. Despite its high bandwidth, it is not widely used in robotic and embedded applications due to its software complexity and lack of providing real-time communication in a scalable manner. Specifically, its usage of CSMA/CD method for medium access control, and packet retransmission and buffering feature of TCP/IP prevents it from satisfying real-time constraints. There are several studies in the literature for real-time communication with Ethernet, such as PROFINET IO [16], TCNet [1] and [17]. In particular, [17] is an interesting study where the authors claim to have used Ethernet as a real-time communication infrastructure in an on-body network for a humanoid robot, HRP-3P [3]. Real-time communication with Ethernet is realized by adopting a communication method where TCP/IP layer is eliminated and the data link layer is directly used for making processing time deterministic; and controlling transmission time using a real-time operating system, ARTLinux [15].

RISEBus is a low-cost real-time network protocol geared towards implementation of distributed control systems [18]. It is used in small mobile robots such as RiSe robot[25]. The RISEBus builds an infrastructure upon which the URB is founded, and consequently, it is explained in more details in Chapter 3.

The *Universal Robot Bus (URB)* is a real-time communication framework that facilitates exchange of data between a master controller (the URB CPU) and slave components (the URB Nodes) of a distributed digital system [5]. The reasons for preferring a distributed architecture instead of adopting a central star topology are explained in Section 1.2, as providing scalability and extensibility, rendering physical implementation easier and reducing noise interference. The shared bus used to connect the nodes of the URB system is an I^2C bus.

It is important to note that instead of a simple master-slave implementation, URB incorporates a gateway between the CPU and the nodes, called the URB Bridge, behaving as a slave to the CPU, but a master of the nodes connected to it. This can be seen more clearly in Figure 1.1, where the arrangement of eight nodes in central and distributed architectures are compared. For the distributed case,

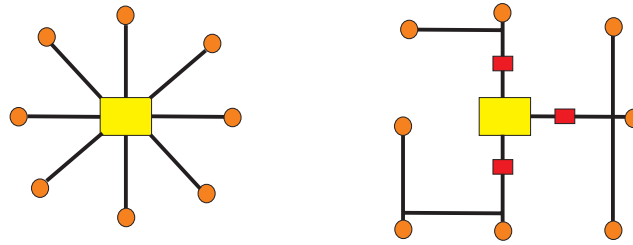


Figure 1.1: Arrangement of eight nodes in central and distributed architectures.

as in the URB, there are three bridges connected to the CPU, and the bridges master the underlying I^2C bus to which nodes are connected. Further details of the URB can be found in Chapter 3.

1.4 Motivation and Contributions

URB is an appropriate distributed control infrastructure for the reactive autonomous robots which we are interested in. Its success in providing an efficient, fast, real-time, synchronous and cheap internal communication within small-scale mobile robots is detailed in [4]. However, we cannot use its full capacity when an uplink connection with a conventional serial communication interface such as RS232 is used due to insufficient channel bandwidth, specifically when the number of nodes attached to the I^2C bus of a bridge increases. This problem motivated us for the incorporation of a high bandwidth uplink channel to the URB framework. The Universal Serial Bus was chosen as the high bandwidth channel, due to its advantages such as high and guaranteed bandwidth, hot-plugging, easy cabling and low-cost physical connectors.

There are two negative aspects of USB that make it not a very common preference for real-time distributed applications. The first one is its lack of support for interrupt mechanisms. The USB Specification does not allow the peripheral device to interrupt the CPU; instead a polling mechanism is used. The second drawback of USB is its non-determinism in terms of timing and synchronization of multiple devices. More specifically, USB has no provision for providing accurate timing information to multiple devices and has very limited capacity to

synchronize events on multiple devices [12].

There are a few studies in literature which claim to incorporate a real-time support within USB architecture. Authors of [12] claim that they developed a highly deterministic distributed control platform which combines the widespread compatibility features of USB with advanced timing and software features required for an industrial PC I/O communication platform. In [10], the authors claim that they invented a method for accurately determining the specific time of occurrence of a real-time event that is monitored by a peripheral USB device. Authors of [32] uses USB as a communication infrastructure of a CNC system, using real-time processing capabilities of RT-Linux for time critical requirements, and USB device drivers to obtain deterministic and real-time communication between the CNC system and peripheral machines. In [30], authors claim to have developed an innovative USB strategy to implement integrated power monitoring applications and that the high-bandwidth USB connectivity will boost real-time capability of their distributed power monitoring system.

These studies mostly define real-time operation as the ability to determine the exact time of occurrence of an event, and base their studies on finding methods for accurate time determination using the USB architecture. However, our requirements for URB are different than this definition of real-time operation. Since USB polls the bus at 1ms intervals at full speed, we are looking for implementing a system where the round-trip latency¹ of the uplink is below 2ms, which is sufficient for us to define our system as real-time. More specifically, the contribution of USB in terms of bandwidth is so much for us that we can tolerate a latency of 2ms, since using USB will eventually increase overall performance of the URB uplink. For instance, [4] finds out that the uplink latency of a RS232 connection for a small sized packet is 0.88ms. Since the bandwidth of RS232 is limited to 230.4Kbps and its latency increases with the size of transfer, a USB uplink with a latency of 2ms and bandwidth of 12Mbps will surely perform better than RS232.

¹Round-trip latency is the time required for a packet to travel from a source to a destination and back.

As a result of these observations, we implemented a USB uplink channel for the URB, which is the primary contribution of this thesis. Hardware components used in the USB uplink implementation as well as our software implementation are covered in detail. These details include the firmware running on the gateway, a Linux based device driver and a client control software that uses a USB library, which is an instantiation of the common URB CPU API for the USB uplink, running on central controller, and finally sub-protocols between the application-driver and driver-firmware layers, all covered in Chapter 4. The thesis also includes our experiments to estimate the performance of the USB uplink in terms of its round-trip latency, bandwidth, scalability, robustness, and reliability, covered in Chapter 5. Finally, this thesis also serves as a reference on distributed systems, device driver development, Linux kernel programming, communication protocols, USB and its usage in real-time applications.

1.5 Outline

The thesis is organized as follows. The first chapter gives a basic introduction to background necessary to digest the contributions of this thesis, including brief information about the domain of the thesis, the purpose of the proposed work, and previous studies from related literature. The second chapter presents basic background knowledge required for a clear understanding of the URB framework details, such as I/O subsystem, device driver development in the Linux operating system and basics of the Universal Serial Bus Protocol. Chapter 3 explains the details of the URB framework. The fourth chapter gives details of the USB uplink channel for the URB, which constitutes the primary contribution of this thesis. Chapter 5 presents a model that is suitable for the analysis of USB throughput characteristics, and then explains details of conducted experiments on the USB uplink for the URB, as well as presenting their results and a discussion on what those results mean. The last chapter gives a final overview of the proposed solution, as well as what is expected to be done as future work.

Chapter 2

Background

The communication framework presented in this thesis involves the integration of various hardware and software components. This chapter deals with the basic background knowledge required for a clear understanding of the URB framework details. The topics covered in this section are quite complex, and it's not possible to cover all details. Therefore, only brief summaries of the topics are presented. We first briefly overview I/O management alternatives within a computer system in Section 2.1. Secondly, general features of a device driver, a crucial part of any I/O management subsystem, are presented in Section 2.2. Next, the details of device driver development for Linux systems are explained in Section 2.3, including primary challenges such as concurrency management, synchronous operations, and details of character device drivers. Following that, the Universal Serial Bus (USB) protocol is summarized in Section 2.4. Then, USB device driver development for Linux platforms is briefly explained in Section 2.5. Finally, the chapter ends with a discussion on the USB module of the Silabs F340 board, which is used as the experimental platform for our URB implementation. Most of this chapter is based on information compiled from [8, 23, 27, 29].

2.1 I/O with Peripheral Devices

The core of a computer system is generally considered to be composed of a CPU and memory. Therefore, any other device connected to this CPU-memory pair, such as disks, mice, keyboards, monitors and so forth, can be considered as an input/output (I/O) device. I/O devices can be categorized as block, character and network devices [23]. Block devices transfer data in fixed-size blocks and each block is addressable, allowing seek operations [29]. Storage media such as disks and tapes are the most common block devices. On the other hand, character devices transfer a stream of characters, and are not addressable, therefore can be accessed only sequentially. Network devices allow packet transmission across network interfaces. Although some network connections, i.e. TCP, are stream-oriented as the character devices, packet transmission is handled differently by the network subsystem of the operating system.

I/O devices typically consist of an electronic component, called the *device controller* and mechanical components, which is the device itself [29]. The controller's job is to abstract physical details of data representation within the device and present a logical view to the operating system. For example, a disk controller converts the serial bit stream extracted from a specific cylinder, track, sector triple of a disk hardware into fixed size logical blocks, and sends the blocks to the related subsystem of the operating system, to the file system in this case. The level of this abstraction may change according to the device controller, offloading some of the conversion between physical and logical representations to the operating system. Moreover, the controller communicates with the device through a standard interface, i.e. SCSI, IDE, which allows identical devices to be handled by the same controller.

I/O operations can be handled using different approaches by the hardware. In *Programmed I/O*, the CPU is in charge of doing all the I/O operations using a polling (busy waiting) mechanism, where the execution loops until the I/O transmission ends. Since I/O operations are much slower than transmissions between the CPU and the memory, this method is not efficient and *Interrupt-Driven I/O*

is used to prevent the wasting of CPU cycles while waiting for the I/O operation to complete. In this approach, the CPU is scheduled for another task as soon as an I/O operation is initiated, and the device generates an interrupt upon finishing the processing of the current byte/word, which in turn invokes the execution of the associated interrupt service routine (ISR). The last approach is *I/O using Direct Memory Access (DMA)*, which tries to improve I/O performance by reducing the number of context switches that are generated from the continuous interrupts of Interrupt-Driven mechanism. The DMA controller has access to the system bus independent of the CPU, and regulates data transfer between the memory and device controllers without CPU intervention, optimizing system performance. The three approaches have their own advantages and disadvantages, and any of them may outperform the others depending on the situation in which they are used [29].

2.2 Device Drivers

A device driver is a kernel program that implements device-specific aspects of generic I/O operations, allowing high level programs to interact with a device. Device drivers force hardware to respond to a well-defined internal programming interface; completely hiding the details of how the device works. User activities are performed by means of a set of standardized calls, i.e. *open*, *close*, *read*, *write* etc., that are independent of the specific driver. Mapping those calls to device-specific operations that act on real hardware is the role of device driver [23].

This role can be clarified by viewing Figure 2.1, where the position of device drivers among the layers of an I/O system is shown. Here, a user process invokes the I/O request by a call from the standard user libraries of the language, i.e. **fread**, and execution is then switched from user space to the kernel space (See Section 2.3 for details). The device-independent I/O subsystem of the operating system performs all necessary actions such as checking the cache if the data to be retrieved is there, arrange parameters etc, and then invokes the I/O system

call, i.e. read. The read system call for that specific device is implemented in the driver of that device, where the appropriate I/O operation is initiated by writing associated commands to the controller of the device. The driver remains blocked until the completion of the I/O. When the device controller completes its task, e.g. sending the character to the driver, the device generates an interrupt and corresponding ISR extracts the status from the device and wakes up the sleeping process in order to finish off the I/O request and let the user process continue [29].

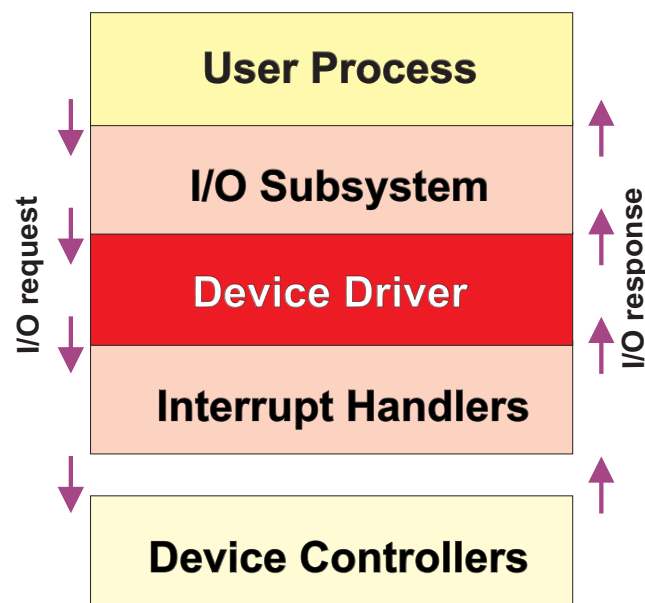


Figure 2.1: Typical layered decomposition of an I/O system.

Device drivers are usually device-specific and implemented by the manufacturers of the corresponding device. Although the duties of each driver depends on the device they are developed for, there are common functions which almost all drivers are supposed to provide. The most obvious one is to control the device by writing the commands to the appropriate register of device controller, and read status information. Data reading and writing is naturally a fundamental part of driver functionality. There are also other functions of drivers such as device initialization and shut down, power management and event logging [29].

Since the driver provides an abstraction of the device to the user processes, there should be some design considerations on the level of abstraction and the

driver functionality. In the first place, a device driver should be policy free, dealing with making the hardware available, leaving all the issues about how to use the hardware to applications [23]. *Policy* is a widely used term in operating systems terminology in conjunction with the term *mechanism*, and the distinction between the two is an important basis behind the UNIX design. While the mechanism defines the capabilities that are provided by the system, the policy determines how those capabilities can be used. For example, a disk driver is expected to provide a mechanism for disk access by showing the disk as a continuous array of data bytes to the user application, regardless of any physical details of a disk. On the other hand, user libraries of the driver may provide policies depending on the requirements of the user such as managing the user permissions for accessing disk file system. Although different libraries may implement different policies for the same driver, the way that those libraries access the disk is the same, which is using generic system calls the operating system provides to the user. This uniform interfacing for device drivers by the operating system allows easy development of device drivers, since a programmer knows the common functionality that is expected to be implemented, and the kernel functions that are allowed to be called from the driver. The concept of mechanism-policy provides a great flexibility to the system, since different requirements could easily be satisfied by only implementing new policies, without ever touching the mechanism.

The second important aspect of a driver is the management of concurrency, which arises when multiple applications or driver threads use the same device at the same time, or the target user applications are multi-threaded. The developer of the driver has to choose the best method for concurrency management by considering performance constraints and functional requirements of the driver.

Thirdly, a driver needs to have a buffer management mechanism in order to send and receive data in an efficient manner. The buffering mechanism plays an important role in the communication performance between the device and the application, and includes design constraints such as determining the kernel buffers to be used for storing data temporarily and the buffer sizes, whether static or dynamical allocation will be used and so on. Several factors play role upon choosing

a buffering mechanism such as the type of device, i.e. whether the target is a character, block or network device, timing and performance constraints of common user applications, and the hardware capabilities, i.e. maximum bandwidth the device can provide. *Double buffering* is a widely used mechanism, where there are two kernel buffers on the receiver/or sender, and while one buffer is accumulating input, or sending data, the other buffer is busy with copying the data to/from the user process [29].

Finally, the security of the system is a fundamental issue since poorly written device drivers are one of the most common security holes for the hackers. Device drivers are usually not expected to possess a specific security policy, since policies are rather implemented up in the higher levels of the kernel. However, there are some general security issues that the driver implementers need to be aware of. To begin with, the driver should be written carefully to avoid common security bugs such as the buffer overrun vulnerability. Next, any device access methods that could damage the system when improperly used, i.e. loading a firmware, setting an interrupt line or default buffer size, should be given to privileged users by the driver. What is more, any driver that takes input from the user, or decodes and interprets data sent by the user should make sure that unexpected data from the user does not compromise the system. Lastly, uninitialized memory should be avoided considering that it is a common resource of information leakage.

Other design issues of drivers include the support for synchronous and asynchronous operations, the ability to be opened multiple times concurrently, error reporting and exploiting full capabilities of the hardware. All of these items belong to general design considerations and the list can be extended to fulfill other requirements of communication.

2.3 Linux Kernel Programming

UNIX based operating systems provide two levels of operating modes in order to protect the system resources from unauthorized access. These operating modes

are the *supervisor mode*, in which all computer instructions including privileged ones are allowed to be executed, and the *user mode*, where the usage of instructions and system resources is limited [23]. User applications run in user space, or the user mode, whereas subsystems of the operating system, i.e. task or memory management, I/O control, run in kernel space, the supervisor mode. The subsystems of the kernel are supported by various low level software, and most of this kernel software is made up of kernel modules which are pieces of code that can be loaded and unloaded into the kernel upon demand. Kernel modules extend the functionality of the kernel without the need to reboot or recompile the system. Regarding the wide spread of personal computers usage in recent years, it is rather suitable to use kernel modules for the development of device drivers, which makes the module installation and deployment easier for end users.

Since device drivers accomplish their duties by linking to the operating system and they can be implemented as modules, they surely run in kernel space, and have different characteristics than user applications. One of the differences between a kernel module and a user application is that while a module can only call the functions of kernel for memory management, DMA control, timer management and so on, an application can call any function in user libraries. As a consequence, a kernel programmer has much more limited resources as previously implemented libraries than a user application programmer. Another difference is that while a module is event-driven, which means the module registers itself to the kernel with an initialization routine so as to serve future requests of other applications, a user application does not necessarily have to be event-driven, in fact, most of them are procedural. Also, a module should have a carefully implemented cleanup routine in which every allocated resource is released since the resources it allocates remain in the system until rebooting. Lastly, it is easy to develop and debug code in user space, since there are many integrated development environments available. However, there are many difficulties in kernel code development including the lack of an advanced debugging support, the need for a much complex compiling and linking process using make programming techniques, the necessity of inserting kernel version and platform dependency checks using preprocessor directives, and the fact that kernel faults may lead to a crash

of the whole system.

As it can be inferred from the aforementioned characteristics of a device driver, kernel programming has many challenges that need to be carefully considered. In this section, basic concepts of programming linux kernel modules for device driver development, including character devices, concurrency management, and synchronous I/O are discussed. Kernel programming concepts such as the compiling, loading and unloading of kernel modules, error handling mechanisms for module initialization failures, exporting module symbols to the kernel symbol table, and other module programming details that are specific to Linux kernel are behind the scope of this thesis, and further reading for these issues can be found in [23].

2.3.1 Character Drivers

Character devices have the capability of transferring any number of data bytes from/to user processes. As opposed to block devices, which operate at fixed size blocks for data transfer, character devices are more common and implementation of device drivers for them is somewhat easier.

Character devices are accessed through device files, also known as special files or nodes in Linux terminology, usually located in `/dev` directory. A single character device, in fact any device, is uniquely identified by a pair of major/minor numbers. The major number identifies the driver associated with the device file. Therefore, I/O control for devices sharing a major number are handled by the same driver. The minor number is used by the driver to differentiate which device it's operating on, in case the driver handles more than one device. In other words, device files with the same major number are uniquely identified by their minor numbers.

The first step for setting up a character device is to obtain the device major and minor numbers that the driver is going to use. Linux kernel provides functions that can be used to obtain device number either statically, in which the user is

responsible for selecting unused device numbers and pass them as parameters to the function, or dynamically where the kernel allocates the numbers itself from its list of unused major numbers.

Following the allocation of device number, the character device driver is expected to register itself with the Linux I/O subsystem, which is a fundamental requirement of event-driven programming as mentioned in Section 2.3. The registration includes the process of giving the I/O subsystem information about which devices the driver supports and which driver functions to call when a device supported by the driver is inserted or removed from the system. Various alternative methods exist for registering the driver module to the Linux kernel in the initialization function, i.e. `init()`, all of which do the same registration process using different kernel implementations. All actions taken in the registration processes should be rolled back in the cleanup functions of the driver.

Once the driver is successfully loaded, and the device is registered including the completion of necessary initializations, the character device is ready to communicate with any user application by means of system calls. A *system call* is a function that can be called by any authorized user application in order to use services provided by the operating system. System calls are usually used for situations where it is not suitable to give the user level process permissions for direct control over the system, such as I/O with peripheral devices or any form of communication with other processes. In Linux, system calls are implemented in the driver according to the requirements of the communication protocols between the computer system and device. For the implementation of system calls, Linux kernel provides a commonly used structure, `struct file_operations`, in `<linux/fs.h>`, which is a collection of function pointers as in the following example declaration.

```
static struct file_operations my_device_fops = {  
    .owner    =    THIS_MODULE,  
    .open     =    my_device_open,  
    .release  =    my_device_release,  
    .read     =    my_device_read,  
    .write    =    my_device_write,
```

```
        .ioctl =      my_device_ioctl,  
};
```

This declaration simply tells that the associated driver has the implementations of open, release, read, write and ioctl system calls which the user application may use to communicate with the device. This example contains only the most commonly used file operations, and there are a number of other system calls which can be invoked by the application. Any function pointers which don't take place in the declaration are unsupported operations. The exact behavior of the kernel when an unsupported operation is invoked by the application is different for each function and further details for each function can be found in [23].

File operations defined above must be associated with the driver in order to make sure that the kernel successfully invokes the correct function to provide a reliable communication between the character device and the user application. This association is realized by using the kernel structure `struct file` defined in `<linux/fs.h>`. This structure represents an open file in the kernel and it should not be mixed with the `FILE` structure defined in C library, which is a user space pointer for the open files. The file structure has a field called `f_op` which is set to the above declared file operations structure in order to establish the association between the device file and the operations implemented by the driver. It is important to be aware that the `file` structure represents an abstract open kernel file descriptor, not a file on a disk. The kernel uses another structure called `struct inode` in order to represent files internally. There can be numerous `file` structures representing multiple open descriptors on a single file, but they all point to a single `inode` structure.

It is also worth explaining what the above defined file operations actually correspond to. The `open` function must be called in order to start a communication with the device and it is invoked for a driver to process any initialization including the determination of which device is being opened, checking device-specific errors, initializing the device if it is being opened for the first time and so on. The `release` function is the complementary clean-up utility, also referred to as the close function. It deallocates anything `open` has allocated and shuts down

the device on the last close. The most important functions for establishing the communication are `read` and `write` calls. The `ioctl` function offers a way to issue device-specific commands that cannot be achieved with `read` and `write` calls such as formatting a disk.

It is essential to elaborate the `read` and `write` calls since they form the basis of the communication between the user application and the device. Data retrieval from the device is realized with the `read` call, whereas data is sent to device with `write`. Here are the prototypes of the `read` and `write` functions:

```
ssize_t read(struct file *filp, const char __user *buff,  
             size_t count, loff_t *offp);
```

```
ssize_t write(struct file *filp, const char __user *buff,  
             size_t count, loff_t *offp);
```

For both methods, `filp` is the file pointer and `count` is the size of the requested data transfer. The `buff` argument points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed. Finally, `offp` is a pointer to a long offset type object that indicates the file position the user is accessing. The return value is a signed size type, containing the number of bytes successfully transferred, or an error value if transfer is unsuccessful. It might also have a value of zero indicating there were no data to read or write due to reaching end file.

It is important to note that `buff` is a user space pointer and therefore cannot be dereferenced by the kernel code. This is due to various reasons such as that the user space pointer may not be valid in kernel mode, or the required user space memory page may not be resident in memory and other security related reasons.

Since it is not suitable to dereference the user buffer, Linux kernel provides two functions to transfer data between the user and kernel spaces, which are `copy_to_user` to transfer data to user space and `copy_from_user` to transfer data to kernel space. These functions must be used with care on the grounds

that they access user space. Particularly, they must be called from non-interrupt context where the process can securely sleep and reentrant context where other driver functions can concurrently execute.

2.3.2 Concurrency Management

Concurrency is a property of systems in which several computational processes are executing at the same time, and potentially interacting with each other [22]. Concurrent processes may be executing truly simultaneously, in the case that they run on separate processors in a multi-core system, or their execution steps may be interleaved to produce the appearance of concurrency, as in the case of separate processes running on a multitasking system.

The difference between a sequential system and a concurrent system is the fact that the processes which make up a concurrent system can interact with each other [22]. Concurrent use of shared resources is the source of many difficulties. Race conditions involving shared resources can result in unpredictable system behavior. The introduction of mutual exclusion can prevent race conditions, but can lead to problems such as deadlock and starvation [22]. Correct sequencing of the interactions between different tasks, and the coordination of access to resources that are shared between tasks, are key concerns for concurrent programming.

In a modern Linux system, there are numerous sources of concurrency and, therefore, possible race conditions [23]. SMP systems, preemptive nature of the multi-tasking operating system, asynchronous events such as hardware and software interrupts, kernel mechanisms for delayed execution such as workqueues, tasklets and timers are all examples to sources of concurrency. As a consequence, race conditions are inevitable and all software should be developed considering the concurrency issue in order to function reliably. Concurrency management becomes more crucial when we start dealing with kernel code, since any unpredictable behavior of the kernel might result in serious system failures.

Race conditions are a natural consequence of shared access to computer resources, i.e. shared data and peripheral devices. Fortunately, there are strategies for resolving this problem. In the first place, any software should be designed such that there is a minimum amount of shared resources. However, it is obvious that there will still be some shared data since computer system resources are limited. Another, yet the most useful, strategy is that access to the critical section of a program, the code segment in which there is an access to a shared resource, should be controlled by a locking mechanism, making sure only one execution thread can manipulate it at any time.

Linux kernel provides various primitives for different concurrency management schemes. The most commonly used primitive is the *semaphore*, which restricts the number of simultaneous accessors to the critical section of an execution up to a maximum number. When the maximum number is one, which means only one thread is allowed to access the critical section, semaphores are referred to as *mutexes*, which is short for mutual exclusion. In simple terms, semaphores are single integer values and threads request access to a resource by decrementing the semaphore, and signal that they have finished using the resource by incrementing the semaphore. To give an example, you may think of two threads *A* and *B* trying to access a resource *R*, and *R* is protected by a mutex, hence its initial value is 1. When thread *A* accesses the shared resource *R*, *A* holds the mutex by decrementing its value to 0. If thread *B* tries to access the resource *R* while *A* is still holding the mutex, thread *B* is blocked until thread *A* signals by releasing the mutex. The blocking mechanism might be different for various Linux semaphore implementations, but usually thread *B* is put to sleep.

Although semaphores are very powerful concurrency management tools, they cannot be used in an interrupt context because the kernel does not allow code in interrupt context to sleep. The reason behind this kernel restriction can be clarified by a simple example. Consider the situation where an interrupt handler needs to access a resource protected by a semaphore, and that semaphore is already held by another thread. The interrupt handler is expected to be put into sleep and another process is to be scheduled to the processor. However, an interrupt cannot be rescheduled since there is no backing process context for an

interrupt, therefore there is nothing to wake up the interrupt. Therefore we need an alternative method to avoid using semaphores in interrupt context.

The alternative method is to use the *spinlock* mechanism, which is a lock where the thread simply waits in a loop, repeatedly spinning, checking until the lock becomes available. As the thread remains active but does not perform any useful task, the use of such a lock is a kind of busy waiting. Once acquired, spinlocks will usually be held until they are explicitly released, although in some implementations they may be automatically released if the thread blocks, or goes to sleep. Spinlocks are efficient if threads are only likely to be blocked for a short period of time, as they avoid overhead from operating system process rescheduling or context switching, which is the case that happens when semaphores are used. However, spinlocks become wasteful if held for longer durations, preventing other threads from running. The longer a lock is held by a thread, the greater the risk that it will be interrupted by the operating system scheduler while holding the lock. If this happens, other threads will be left spinning, in other words repeatedly trying to acquire the lock, despite the fact that the thread holding the lock is not making progress towards releasing it. This is especially true on a single-processor system, where each waiting thread of the same priority is likely to waste its entire allocated timeslice spinning until the thread that holds the lock is finally rescheduled. There is a much worse scenario where an interrupt handler starts spinning waiting for a spinlock to be released and the thread that held the spinlock also executes on the same processor. In this situation, the non-interrupt code would not be able to release the lock and the interrupt handler would spin forever since interrupts cannot be suspended. This worst case scenario also applies to uniprocessor systems. To avoid this situation, there are variants of spinlocks to be used especially in interrupt context which disable interrupts on the local processor so that the code cannot be interrupted while holding a spinlock.

Although semaphores and spinlocks are the most frequently used concurrency management mechanisms, there are others such as *seqlocks*, *completions*, *atomic variables*, *RCU locks* etc. In this section, a summary of only the concurrency issues which are used in our research is given. Other details of concurrency

management are behind the scope of this thesis and further information on other mechanisms and concurrency policies can be found in [6, 22, 23].

2.3.3 Synchronous I/O

The details of character devices were explained in Section 2.3.1, and read and write calls were emphasized as the most commonly used operations on a device file. When there is a read/write request, the kernel may not satisfy it immediately since there could be no data available in a read request or the output buffer could be full for a write request. In such situations, busy waiting is not suitable regarding performance constraints since I/O operations last relatively longer than kernel operations. The solution is to block the process by putting it into sleep and wake it up when the reasons for blocking no longer exist, which is called *synchronous I/O*.

When a process is put into sleep, it is removed from scheduler's ready queue until an event to put it back into the queue happens. Linux provides functions which can be used easily to put the process into sleep and to wake it up. However there are some rules that should be considered before using these functions [23]. First of all, any call to sleep should never be in *atomic context*, a state where multiple steps must be performed without any sort of concurrent access. Therefore, the kernel code cannot sleep while either the interrupts are disabled or a spinlock, seqlock, or RCU lock is held. It is legal to sleep while holding a semaphore, but it should be kept in mind that if code sleeps while holding a semaphore, any other thread waiting for that semaphore also sleeps. So any sleeps that happen while holding semaphores should be short, and by holding the semaphore, the process that will eventually wake the sleeping one up should never have been blocked. Secondly, since the state of the system could have changed while the process is sleeping, the first thing the process upon waking up should be to check the condition it was waiting for is really true now, and go back to sleep if it is false. Lastly, it must be ensured that there is really another thread of execution to wake up the sleeping one, and that thread is able to find the sleeping process. This last requirement of accessing a sleeping process is implemented by Linux

kernel through a structure called the `wait_queue`.

The `wait_queue` is a queue of processes all waiting for a specific event. A process is put to sleep with the macro `wait_event(queue, condition)` defined in `<linux/wait.h>`, and there are other variants of this macro available. Here the process waits on the queue `queue` until the condition `condition` becomes true. The other process wakes up all the sleeping processes on the queue by calling the function `wake_up(wait_queue_head_t *queuep)` where `queuep` is a pointer to the wait queue `queue`.

The default behavior of the kernel upon a read or write system call to block the calling process until the kernel is ready to receive or send data, which is the synchronous, also known as *blocking*, I/O mechanism. However, kernel also allows *non-blocking*, or *asynchronous*, I/O in which read/write calls immediately return to the application. Consequently, the application should either poll or use more advanced techniques such as signaling, callback functions etc, until the requested number of bytes actually return to it.

2.4 The Universal Serial Bus

The Universal Serial Bus (USB) is a bus architecture that facilitates data exchange between a host computer and peripheral devices. It allows peripheral devices to be attached, configured, used and detached while the host and the peripherals are in operation. USB was developed with the intention of replacing a wide range of slow and different buses with a single bus which all devices could connect to. The USB architecture has grown beyond this initial goal, and now supports high speed communications at 480Mbps, with other advantages such as the support for hot swapping and plug and play, a wide range of guaranteed bandwidth, a large number of dynamically attachable external peripherals, low cost cables and connectors and so on [11, 23].

The USB is strictly hierarchical and it is controlled by one host. The host uses a master/slave protocol to communicate with attached USB devices. This

means that every kind of communication is initiated by the host and devices cannot establish any direct connection to other devices. This seems to be a drawback in comparison to other bus architectures, but since USB was designed as a compromise of costs and performance, this disadvantage can be tolerated. The master/slave protocol solves implicitly problems like collision avoidance or distributed bus arbitration [11].

In this section, an overview of USB key concepts that are essential to understand the rest of this thesis is provided. The details of this overview can be found in USB 2.0 Specification [8].

2.4.1 Bus Topology

A USB system consists of a *USB Host*, where the host controller, client software and device drivers reside for controlling the system, a *USB Hub* which is a special USB device that provides attachment points for other USB devices via *ports*, and a *USB Function*, which is a peripheral device that communicates over the bus and provides a functionality to the host. Devices on a USB system are physically connected to the host via a tiered star topology, which can be seen in Figure 2.2. In this arrangement, the host comprises an embedded root hub, which provides attachment for remaining *external hubs*.

Although the physical connection between the devices and the host is a tiered star topology, the logical connection between the host and the devices is as illustrated in Figure 2.3, where all the logical devices, including the hubs, are connected to the host directly. While most devices use this logical perspective, the host is also aware of the physical topology, so that the removal of hubs is successfully managed by removing the devices connected to it from the logical view. It is important to note that a client software on the host which manipulates a specific device is logically perceived by the USB host such that the client software deals with only the interface it's implemented for, and is totally independent of other devices on the bus.

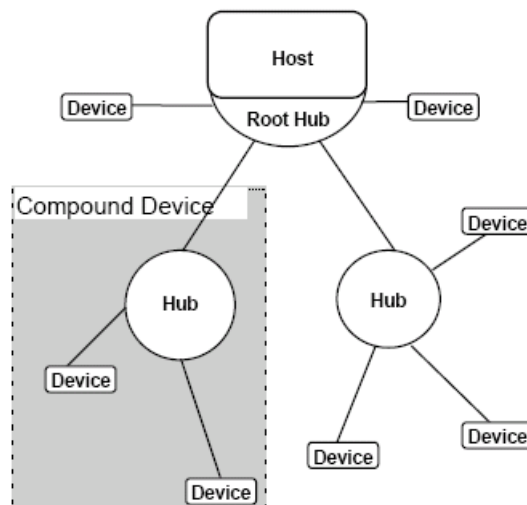


Figure 2.2: Physical Bus Topology of USB [8].

2.4.2 Communication Flow

A system connected via USB has several functional layers, and the communication between a USB Host and a USB device can be modeled as in Figure 2.4, where the black arrows show physical interprocess information flow and the gray arrows show logical information flow between the corresponding layers of the host and the device. A USB Host consists of three components: a USB Client, a USB Subsystem, and a USB Host Controller. The upper component, the Client, consists of all software that is serviced by the USB Subsystem below, including the user level software which interacts with USB devices by sending I/O requests in the form of *I/O Request Packets (IRP)* and receiving responses [8], as well as kernel software such as the I/O Subsystem and the device specific USB driver. Details regarding interactions between the layers of an I/O system are given in Section 2.2, and USB device drivers are further elaborated in Section 2.5.

The corresponding logical layer to the host client on the peripheral device is the *USB Function*, consisting of a collection of interfaces. An *Interface* represents a basic functionality, and handles only one type of USB logical connection. A device may have more than one interfaces, such as a printer with faxing and scanning capabilities. The logical connection between a client software and a function interface is established by bundle of pipes. A *pipe* is a logical data

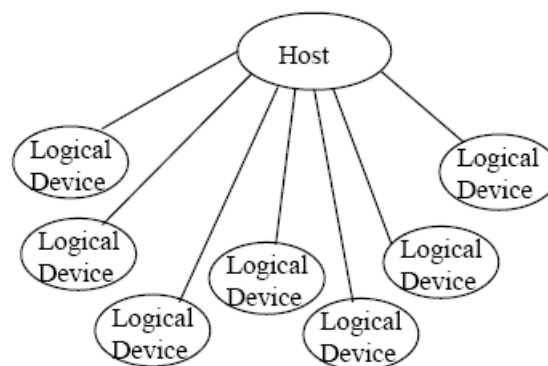


Figure 2.3: Logical Topology of USB [8].

channel between the client and a particular endpoint on the device, where an *endpoint* represents a logical data sink of a USB device.

The middle component of the host, the USB Subsystem, consists of the Host Software (HS), the USB Driver (USBD), the Host Controller Driver (HCD) and the operating system specific HCD Interface (HCDI) between the HCD and USBD, as can be seen in Figure 2.5. The HS consists of various USB Driver (USBD) clients such as the hub driver and other platform specific custom drivers. The USBD provides an abstraction of the device to upper level clients, mapping requests from many clients to the below appropriate Host Controller Driver (HCD), as well as providing a collection of mechanisms that the device drivers use to access USB devices. The USBD is also involved in device configuration by accepting or rejecting a bus request upon device attachment considering the bandwidth requirements of the device, and provides data transfer mechanisms for the IRPs. The HCD maps various HC implementations into the USB Subsystem and provides an abstraction of the HC hardware to the upper levels. As a whole, the USB Subsystem manages USB resources such as bandwidth and bus power, and performs the translations between the client data structures, hence the IRPs, and the USB transactions, which involves addition of USB protocol wrappers to raw IRPs. A *USB transaction* is a stage of data transfer process, and typically

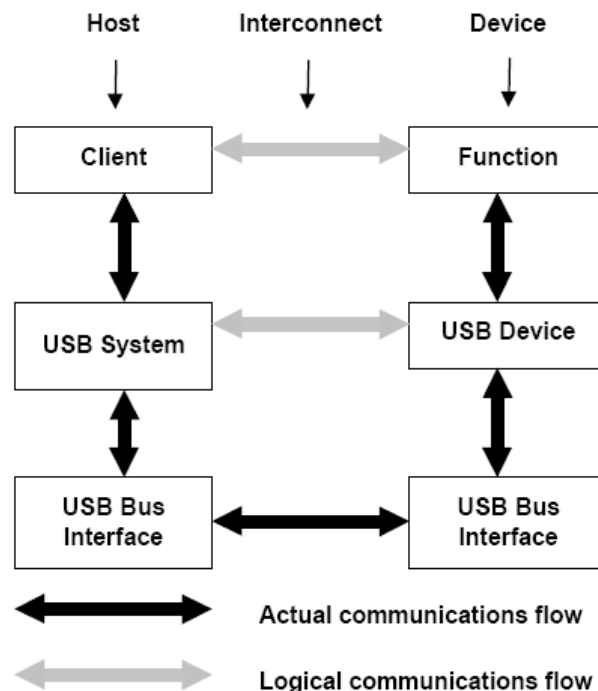


Figure 2.4: Interlayer Communications Model for the USB standard [8].

consists of three packets, token, data and handshake, which are transmitted between the host and the device. Therefore, the transfer of an IRP involves the transmission of one or more transactions, and more details can be found in 2.4.4.

The lowest layer, the USB Host Controller (HC), is the interface between the host and the device, and is in charge of translating the transactions into packets, transferring the packets between the upper USB Subsystem and the USB device controller on the device side, serializing/deserializing through its Serial Interface Engine (SIE) subcomponent, handling transmission errors, providing attachment points to the USB wire through its integrated root hub and generating frames or microframes. While *frame* is a 1 millisecond time base established by the USB protocol on full/low speed devices, *microframe* 125 microsecond time base for high speed devices.

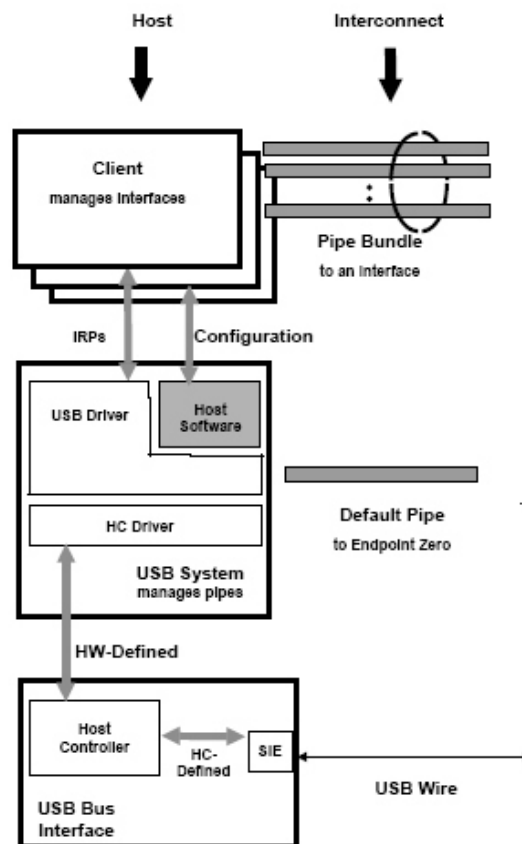


Figure 2.5: Internal Layout of a USB Host [8].

2.4.3 Data Transfer Types

Although all USB devices need to communicate with their hosts in order to provide functionality, the communication flow requirements of various devices can be different. As a consequence, the USB specification defines four types of data exchange between a host and a device: Control, Interrupt, Isochronous and Bulk Transfers.

The Control Transfer is a bidirectional type of data flow used for configuration/command/status type communication between the host and the device. The configuration is established at attachment time of the device using the *Default Control Pipe*, which is a pipe consisting of two endpoints and assigned the endpoint number zero. This pipe provides access to the configuration, status, and control information of the USB device by the USB subsystem of the host as soon

as the device is powered and reset. Although any USB device is required to support the default control pipe, a device can also provide endpoints for additional control pipes for its own implementation needs.

The Interrupt Transfer mode is used for the asynchronous transmission of small amounts of data, typically in devices such as keyboards and mice. When the host requests an interrupt transfer, the data is transferred between the host and the device at constant intervals, between 1ms and 255ms, respecting a guaranteed maximum service period.

The Isochronous Transfer mode is used for the exchange of time sensitive information, such as an audio or video stream, where keeping a constant stream of data in large amounts is more important than data delivery guarantee. Real-time data collections such as multimedia sources use this type of transfer.

The Bulk Transfer mode is used in devices that need to transmit large amounts of data using full available bus bandwidth in a reliable manner. The USB protocol does not give any guarantee on latency for this type of transfer. Any packets exceeding the maximum bulk packet size are automatically split and transferred accordingly. The printers, scanners, network and storage devices are typical examples of its usage.

A device is required to use the control pipe for configuration, and free to use any of the other transfer types according to its communication needs. The USB protocol provides the separation of different communication flows to a USB function so that a more efficient bus utilization is achieved. The device can be configured to use different transfer types for its various endpoints, which might be useful for devices with multi-interfaces. Table 2.1 shows a comparison of these four transfer types, in terms of different characteristics of the communication. Supported modes denote transfer modes with different maximum speeds, where the upper limit is 1.5Mbps for low speed (LS) mode, 12Mbps for full speed (FS) mode and finally, 480Mbps for high speed (HS) mode. The pipe type describes communication mode in terms of being stream or message. A stream pipe delivers data in only one direction with no USB-defined structure on data content, whereas a message pipe has a defined USB format and data can flow in both

Transfer Type	Control	Interrupt	Isochronous	Bulk
Supported Modes	all	all	full/high	full/high
Pipe Type	message	stream	stream	stream
Data Size	small	small	large	large
Max Payload at FS (bytes)	64	64	1023	64
Max Payload at HS (bytes)	8-64	1024	1024	512
Overhead at FS (bytes)	45	13	9	13
Bandwidth Priority	medium	high	high	low
Latency Guarantee	no	yes	yes	no
Delivery Guarantee	yes	yes	no	yes
Direction	bi-	uni-	uni-	uni-
Transfer rate	non-periodic	non-periodic	periodic	bursty
Used in	default pipe	keyboard, mouse	multimedia streaming, telephony	printer, network, storage

Table 2.1: USB Transfer Types

directions. The payload is the data portion of a data packet, and the maximum possible payload values for full and high speed devices are given in bytes. The maximum payload size for a specific transfer type is HC implementation dependent, and the values given in the table are only an upper limit for these values. The protocol overhead is the number of bytes in a USB formatted packet that the protocol wrappers of the data payloads consist of. The delivery guarantee means that if a transfer fails due to errors, the transfer is retried until the packet is successfully delivered. The transfer rate can be bursty, meaning that data transfer can be initiated asynchronously, or periodic, where data transfer is established periodically, maintaining a constant rate.

2.4.4 Packet Transmission

The USB protocol defines four types of packets: token, data, handshake, and start of frame (SOF). The token packets determine the sequence of forecoming events, and have three types which are the *In Token* informing the device that the host will receive data, the *Out Token* informing the device that the host will send data, and the *Setup Token* indicating the beginning of a control transfer. Data packets carry their payload in their data fields. Handshake packets are for acknowledgment and error correction, and can be any of the ACK, NAK, or STALL types. Lastly, SOF packets are sent by the host every 1ms.

A USB transfer consists of one or more transactions, where a transaction is initiated by the host and sequenced by the transmission of a token, data and handshake packets respectively. In this sequence, the type and the source of the packets (host/device) depend on the transfer type. Transmission of packets for each transfer type requires the knowledge of low level details, which can be found in [8].

2.4.5 USB Descriptors

When a USB device is attached to a port, it is first powered and then immediately reset by the host controller. When the device is in this default state, the host can only access the default control pipe of the device. The host controller queries the default pipe to retrieve the attributes of the device, which are used for the address assignment and configuration of the device so that other endpoints can be accessed to control and exchange data with the device. The attributes of the device are generally called *descriptors*. The hierarchy of USB descriptors is illustrated in Figure 2.6.

The *device descriptor* resides at the top of the hierarchy, which is unique for a USB device and identifies the device class, device and vendor IDs, number of configurations etc. A device has one or more *configuration descriptors* describing

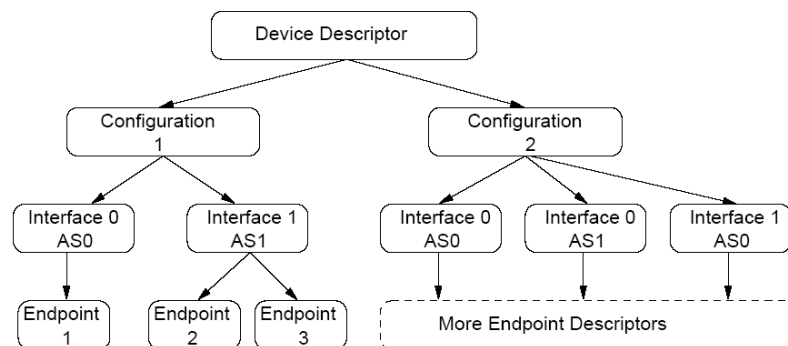


Figure 2.6: USB Descriptor Hierarchy [11].

the power requirements of the device, and number of interfaces. Only one configuration can be enabled at a given time. A configuration has one or more *interface descriptors*, which define a group of endpoints providing a specific functionality. An interface can have *alternate settings* in order to change the settings of the interface on the fly, without effecting the operation of other interfaces of the device. Lastly, an interface descriptor has zero or more *endpoint descriptors* that comprise information corresponding to the endpoint such as the transfer type, direction, maximum packet size and so on.

There is a considerable amount of details about the fields for these descriptors, which can be found in [8].

2.5 USB Device Drivers for Linux

The Linux kernel supports two main types of USB drivers: drivers on a host system and drivers on a device [23]. The USB drivers for a host system, i.e. a PC, control USB devices that are plugged into it, from the hosts point of view. The USB drivers in a device, which are called *USB gadget drivers*, control how that single device looks to the host computer. Gadget drivers are implemented using Linux Gadget API, and are only appropriate for devices that run an embedded operating system, i.e. Embedded Linux. In terms of driver development, this thesis only covers the USB device driver development for a host. Since the device which our host communicates with is a microcontroller without the support for

an embedded operating system, gadget driver development is outside our scope.

USB device drivers lie between the kernel I/O subsystem, which may consist of different subcomponents such as block, net, char, tty, vfs layers, and the USB subsystem, as mentioned in Section 2.4.2. Generic interactions between different layers in an I/O operation are described in Section 2.2, and illustrated in Figure 2.1, which also applies to data exchange between a user application and a device via the USB protocol. The USB device driver involved in such an I/O operation might be either a *class driver*, which is typically supplied by the operating system, or a *custom driver* implemented by the device vendor. Class drivers implement a set of standard operations that member devices of the class can use without the need of a device specific custom driver. Human interface, audio, mass storage, communication, and display are commonly used device classes, and most operating systems support devices belonging to these classes.

A USB device driver serviced by the USB subsystem can be a character, block or network driver according to the type of device it is implemented for. The distinction between these different devices and their corresponding drivers was given in Section 2.1. Block and network drivers are outside the scope of this thesis, so we will focus on character oriented USB device drivers.

USB character drivers are clients of the USB Core layer of the USB subsystem. The term *USB Core* is used by Linux to denote the USB Core component that is mentioned in Section 2.4.2, as well as the USB specification [8]. USB core provides an interface for USB drivers to access and control USB hardware, without having to worry about different types of USB hardware controllers present in the system [23]. All of the general information about device drivers and character drivers given in Sections 2.2 and 2.3.1 apply to USB drivers as well. There are, however, some differences since the USB drivers and character layer of the I/O subsystem are clients of the USB subsystem, where the layers involved in an I/O operation via USB communication is illustrated in Figure 2.7.

Firstly, the interrupt mechanism of USB drivers is quite different. Since the USB architecture does not allow the typical interrupt mechanism of character drivers and uses a polling within frame periods as explained in Section 2.4, USB

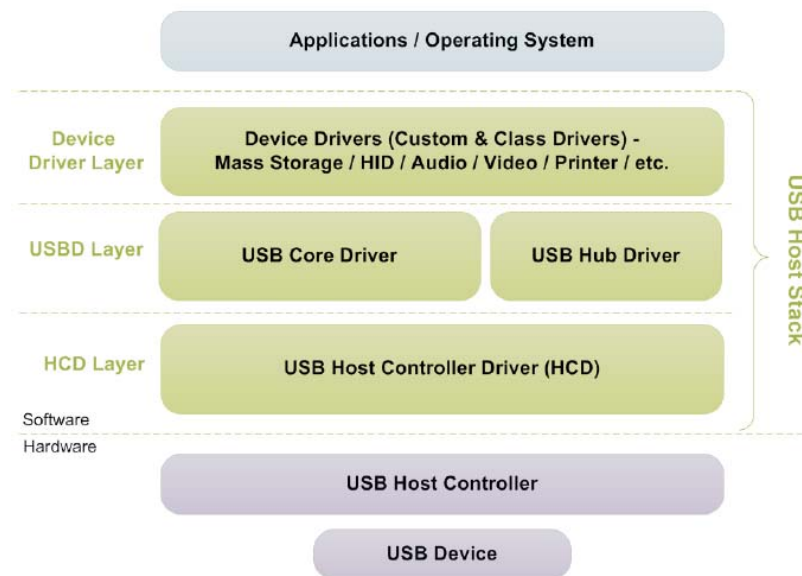


Figure 2.7: General layout of a USB Host Stack [2].

drivers usually do not have explicit IRQ requests or interrupt service routines (ISRs). However, *completion handlers*, also referred to as *callback functions*, provided by the Linux kernel behave like ISRs. These handlers are called by the USB Host Controller Driver (HCD) upon the completion of *USB Request Block (urb)* submission to the device, where a *urb*¹ is an IRP involved in USB communication. Completion handlers run in an interrupt context and hence are subject to restrictions which can be summarized as follows:

- Transfer of data from/to user space is not allowed.
- Anything that may result in suspend of the handler such as calling `wait_event()` function is forbidden.
- Allocation of memory is only allowed with the flag `GFP_ATOMIC`.
- Locking a semaphore is not possible.
- Calling any function that would do scheduling is forbidden.

¹In this document, while the term *urb* written in small letters corresponds to the concept of USB Request Block, *URB* in capital letters refers to the Universal Robot Bus

- The handler should be as short as possible, as any other code running in interrupt context is expected to be. This requirement can be handled by *tasklets* or *workqueues*, which are mechanisms provided by the Linux kernel that split the interrupt handler into two halves, and place the critical sections into the top half, while the rest is left to the bottom half.

The second primary difference is the split of driver setup procedure into two functions, which is a natural result of the hot plugging capability of USB devices. In USB drivers, information on which devices the driver supports is given to the subsystem in the `init()` function of the driver. This is the process of *driver registration*, whereas information on the services the driver presents to the users of the device are given in `probe()` function, which is the process of *device registration*. The sources allocated in the registration processes are given back in the `disconnect()` and `exit()` functions of the driver, where the former is called upon the detachment of the device and the latter is called when the driver module is to be unloaded from the system. The `probe()` and `disconnect()` functions are common in all devices with hot plugging support.

Other parts of the USB drivers are usually identical to other character drivers, including the file operations mentioned in Section 2.3.1, as well as registration and unregistration sequences. USB drivers call the functions and use the structures provided by the USB core when necessary, all of which are defined in `/linux/usb.h`.

Even though there are methods for the exchange of data from a specific endpoint on a USB device that do not require a urb, as in [23], drivers in which the throughput rates is an important consideration are realized by means of urb's. The life cycle of a urb within the driver code can be summarized as follows:

1. A urb is created by a USB device driver (via a call to `usb_alloc_urb()` function in possibly read or write methods of the driver).
2. The allocated urb is assigned to a specific endpoint of a specific USB device (via a call to `usb_fill_xxx_urb()`, where xxx is to be replaced with the preferred transfer type).

3. The initialized urb is submitted to the USB core by the USB device driver (via a call to `usb_submit_urb()`).
4. The urb is submitted to the specific USB host controller driver for the specified device by the USB core.
5. The urb is processed by the USB host controller driver (HCD) that makes a USB transfer to/from the device.
6. When the urb operation is completed either successfully or with an error, the HCD notifies the USB device driver with an interrupt, and the completion handler is scheduled.
7. The status of the transmission is queried and necessary operations are done by the completion handler of the device driver.
8. The urb is destroyed when it's no longer needed by the device driver (via a call to fun `usb_free_urb()`).

Although the exchange of a urb between the host and the device is a complicated process, all steps are abstracted by the USB Core. Therefore, it is sufficient for an implementer to use emphasized functions in an appropriate order with proper parameters, without extensive knowledge about the internal details of the entire process.

2.6 USB module of SiLabs F340 Board

The Silicon Laboratories (Silabs) F340 board is a fully integrated on chip MicroController Unit (MCU) with a 8051-compatible microcontroller core (CIP-51 Core), whose details can be found in [27]. The board has a complete full/low speed USB Function Controller (USB0), which consists of a Serial Interface Engine (SIE) with eight endpoints, USB Transceiver, and 1KB FIFO RAM, which is a commonly used electronic component for data buffering. While the control endpoint, Endpoint0, always functions as a bidirectional IN/OUT endpoint, the others are implemented as three pairs of IN/OUT endpoint pipes.

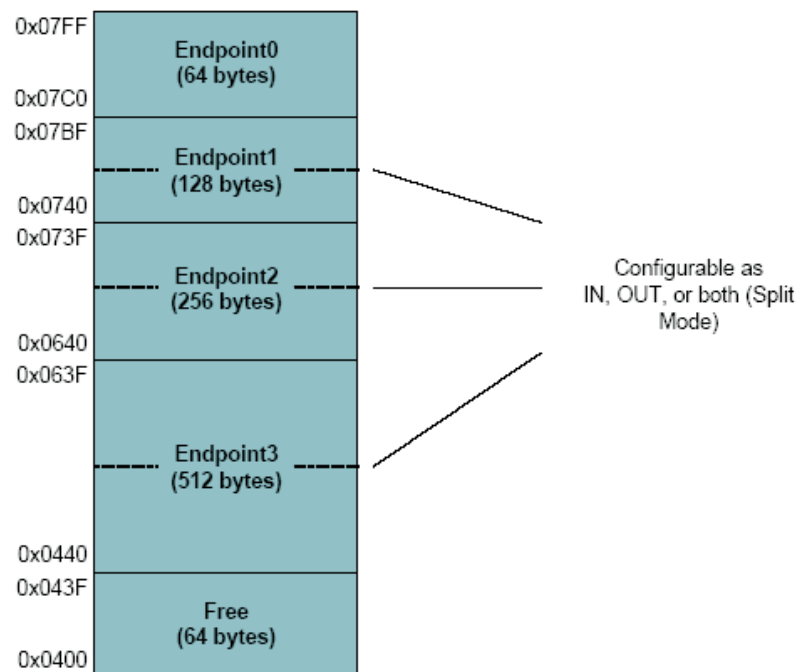


Figure 2.8: USB FIFO allocation for the C8051F340 microcontroller [27].

The assignment of USB FIFO space to endpoint pipes is illustrated in Figure 2.8. As can be seen from the figure, sizes of FIFO space allocated for the endpoints from 0 through 3 are 64, 128, 256 and 512 respectively. Endpoints 1 to 3 can be configured to operate in *Split Mode*, where each FIFO space is divided into two so as to provide an IN and an OUT endpoint FIFO, or to operate with double buffering enabled, as dealt in Section 2.2. In both cases, the sizes of the FIFO spaces are halved, reducing the maximum size of data that can be buffered within a USB frame to half the size. Interested readers should consult the C8051F340 datasheet for more information [27].

Chapter 3

The Universal Robot Bus

The Universal Robot Bus (URB) is a communications framework for distributed digital control systems, facilitating real-time information exchange between a set of physically distributed units, i.e. sensors or actuators, and a central computational unit [5]. The design of URB was inspired by the RISEBus protocol [18].

RISEBus is a low-cost real-time network protocol for distributed control systems. RISEBus specification defines the entire structure of a distributed system including its network structure and firmware architectures of each component. This architecture consists of a central PC connected to an I^2C bus through a gateway called the RISEBus bridge, and hardware nodes connected to the I^2C bus. The RISEBus bridge is the master of the I^2C bus, whereas nodes are slaves. Although the connection between the RISEBus bridge and all nodes is established through I^2C with a half-duplex channel, the PC and the bridge is connected in a full duplex manner with RS232.

URB is a framework inspired from the RISEBus. In this chapter, we give an overview of URB, based on information from [4]. Firstly, a general overview of URB is given in Section 3.1. Secondly, URB components are described in Section 3.2, including the details of the CPU, the bridge and the nodes. Thirdly, the communication model of the URB comprising the uplink and downlink connectivity is presented in Section 3.3. Lastly, URB APIs are explained in Section 3.4.

3.1 URB Overview

The logical view of the framework, perceived by the users of URB, is shown in 3.1.

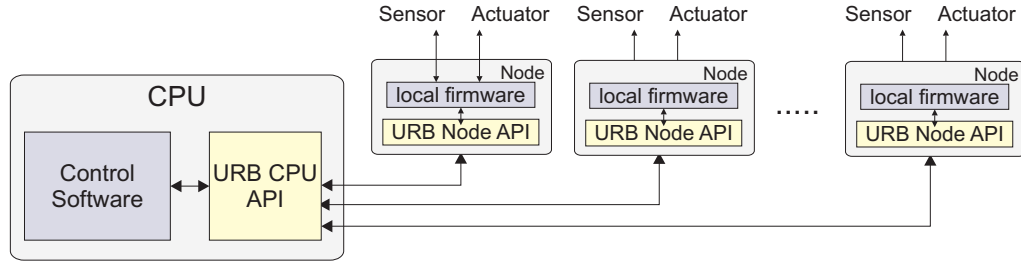


Figure 3.1: Logical Topology of a URB system as seen by the user [4].

Logically, the URB is a system with centralized star topology in which high level control software running on a CPU communicates with local firmware running on sensor and actuator components, referred to as *URB nodes*. This logical structure is implemented by a collection of user-level libraries both for the URB CPU and the URB nodes, and physical implementation details of the framework are hidden from the user. Therefore, both the control software and the local firmware are users of the framework. While a user application on the CPU can simply call functions provided by the *CPU API* in order to communicate with nodes, firmware running on the nodes can call functions in the *Node API* to establish connectivity to the CPU, without knowing any details about how communication is realized by the underlying system.

Direct physical implementation of the logically centralized topology of URB is not appropriate considering our requirements. In the logical view, there is a bottleneck around the URB CPU, so that the performance and scalability of the communication network is limited to the capabilities of the CPU. Since the application domain of the URB is especially small mobile robots, the embedded URB CPU is not expected to be powerful, so it might not be able to provide sufficient throughput as the number of nodes in the star topology increases. Cabling around the CPU might also cause problems with the physical realization of the URB. Therefore, the physical implementation incorporates a gateway called the URB bridge that masters its underlying communication bus and manages requests

from the URB CPU, directed to specific URB nodes in the system. Figure 3.2 shows the physical implementation of the URB with a distributed architecture, where it can be seen that more than one bridge can be connected to the URB CPU with different connectivity options such as USB or RS232. URB nodes are then connected to URB bridges via I^2C buses. The variety of different connectivity options adds another advantage to the framework in terms of extensibility, since different connections between the URB CPU and the URB bridge can easily be added to the system by installing necessary libraries and drivers (Refer to Section 3.3.1 for details).

The connection between the gateway bridge and the CPU is established through an *uplink* channel, whereas nodes are connected to the bridge through a *downlink* channel, all shown in Figure 3.2. Communication between all three components is carried out by means of *transactions*, in which data or command *packets* are transmitted over the channel.

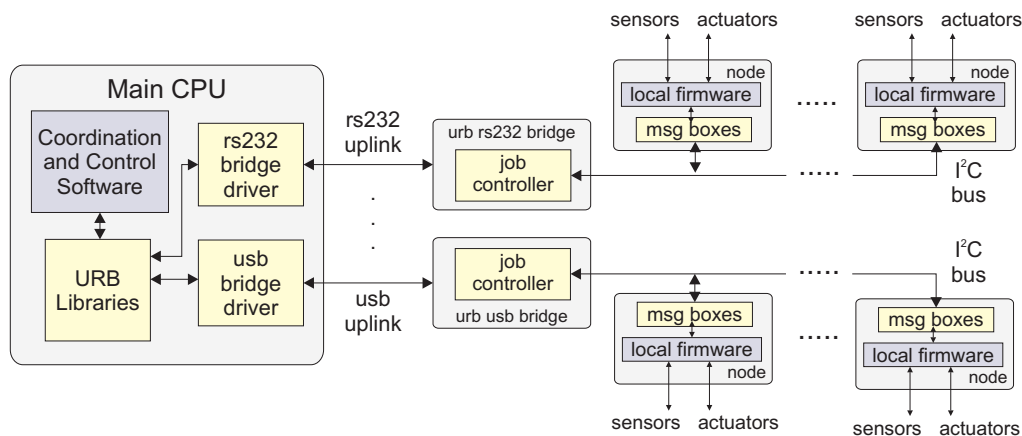


Figure 3.2: Physical Topology of a URB system [4].

3.2 URB Components

A typical URB system consists of three types of devices: the URB CPU (central computational unit) which executes high level control algorithms; multiple URB nodes implementing sensing/actuation with basic signal conditioning; and URB bridges which form gateways between the CPU and sets of hardware nodes [5].

3.2.1 URB Nodes

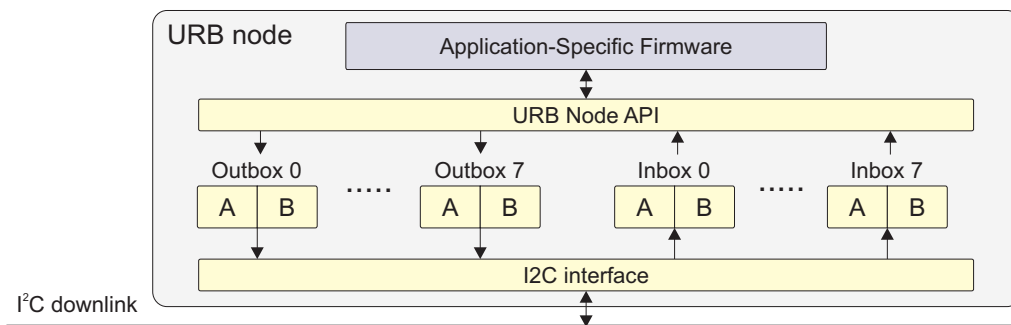


Figure 3.3: Internal software structure of a URB Node [4].

URB nodes are local embedded devices interfacing with sensors and actuators, and accessed through the I^2C bus as slaves. The structure of a node can be seen in Figure 3.3, where node software is composed of an *application-specific firmware*, an *I^2C downlink interface* and the Node API interfacing the two software layers. The application firmware is a user application implemented to deal with the sensory data acquisition and actuator command updates according to the particular needs of the device. Considering a legged-mobile robot as an example, the application firmware for a hip node may be collecting information from the gyroscope sensor attached to a leg, send orientation information coming from this sensor to the CPU using the appropriate Node API calls; then receive actuation responses from the CPU and apply the received commands to the motor shaft, i.e. setting the required motor voltage of the shaft. While this application firmware only deals with the particulars of the devices it's interfacing, the orientation information for gyroscope and motor voltage for motor shaft in this case, it does not have any information about how the data is transmitted to the CPU or received from it. The application only calls the functions provided by the standard URB Node API for establishing the communication.

On the other hand, the downlink interface deals with URB downlink protocol details and basic I^2C communications, and uses a logical abstraction called *message box* while performing data transmission. Protocol details are realized by firstly instantiating a library from the Node API for a specific I^2C compatible microcontroller family, and then handling issues such as message box buffering,

selection of node parameters, prevention of race conditions etc, in this library. By instantiating different libraries from the Node API, rapid development of a wide variety of node designs for different microcontrollers can be established (See Section 3.3.2 for further details).

3.2.2 URB Bridge

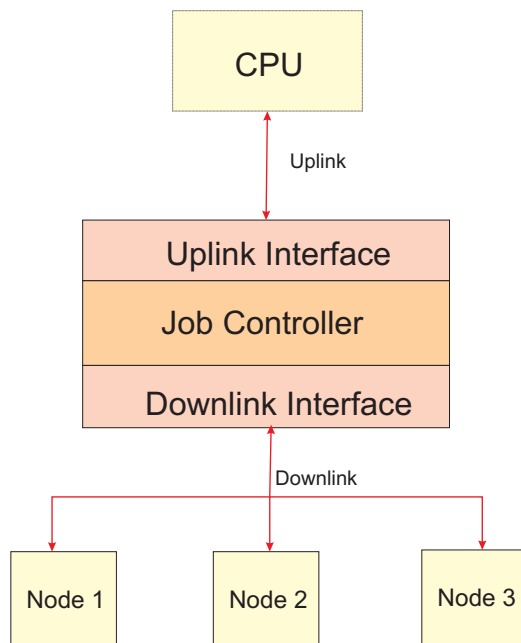


Figure 3.4: The layout of the of the URB Bridge Firmware.

The URB Bridge acts as a gateway that interfaces the CPU with various sensor and actuator nodes in the system. Since it is invisible in the logical topology, users of the URB framework do not really need to know anything about its internals. The software layout of the bridge is displayed in Figure 3.4, where it can be seen that it is composed of three layers: the uplink interface, the job controller and the downlink interface. The uplink interface manages communication of the CPU with the bridge, and its implementation depends on the type of the uplink connectivity, i.e. USB, RS232, PCI, ISA etc. The downlink interface, on the other hand, manages communication between the bridge and the nodes, implementing I^2C /SMBus master functionality [5]. Both uplink and downlink interfaces also deal with necessary buffering of data. The job controller is given many important

responsibilities including providing connectivity, synchronization and scheduling automatic requests.

Connectivity between the uplink and downlink communication channels is achieved by managing CPU requests and responses following the sequence of events below:

1. Incoming requests from the uplink interface are buffered.
2. Requests are processed according to their priority and arrival time. Depending on the type of the request, the job controller may also forward the request to the downlink interface, and receive its response.
3. Responses to completed requests are finally buffered and responses are sent back to the uplink interface synchronously or asynchronously depending on the uplink implementation.

The second duty of the bridge is the arbitration of shared downlink bus access. Since the bridge is assigned as the single master of the I^2C downlink channel, all transactions are initiated by the bridge and the access of slave nodes to the downlink channel also happens under the control and arbitration of the bridge.

Thirdly, the bridge broadcasts a periodic master clock signal for the synchronization of data acquisition across all nodes on the associated downlink channel. Data acquisition within nodes with different clocks may increase data latency. Synchronization of nodes provides periodic data exchange between the bridge and nodes without any overhead on the CPU. As a result, the CPU is spared some workload, and problem of increase in latency is eliminated.

Finally, the bridge schedules *automated data transactions*, which are autonomous requests periodically generated by the bridge. These requests are useful for decreasing the workload of the CPU in situations where the target data is expected to be retrieved periodically from specific nodes, such as the temperature of an industrial system from the thermal sensors, or motor position of a robot. For such requests, the user application simply invokes the automated request

once, and the bridge periodically polls the downlink channel for the appropriate transaction until the request is canceled by the user.

3.2.3 The URB CPU

The URB CPU is the central authority for the implementation of high-level algorithms, coordinating activities for various components in the system. It contacts URB nodes to collect sensory information and applies actuation actions. In order to support this functionality, the CPU toolkit of the URB infrastructure consists of two components: The URB CPU API and libraries, and the URB CPU drivers [5]. The *URB CPU API* is a standard application layer API that allows access to devices in the network as they appear in the logical topology. The details of this API are presented in Section 3.4.1. The *URB CPU drivers* are low level device drivers that handle all details of communication with bridge implementations. Different bridge implementations require different instances of drivers and associated libraries that instantiate the URB CPU API. The USB uplink implementation is the subject of this thesis, and Chapter 4 is completely dedicated to this topic. The USB instantiation library of URB CPU API in Linux is presented in Section 4.4.1, and details for the USB device driver implementation for the Linux platform is given in Section 4.3.

3.3 URB Communication Model

3.3.1 Uplink Communications

The URB framework allows uplink connections from a CPU to multiple bridges. Different connectivity alternatives are possible for uplink connections such as USB, RS232, PCI, ISA and so on (See Figure 3.2, where only currently supported protocols are shown: USB and RS232). Support for a connectivity standard is achieved by installing the necessary device driver capable of communicating with

the bridge uplink interface, and then linking the control software with the appropriate instantiation of the URB CPU API compliant with the installed driver. The interaction between the control software, the driver and the uplink interface firmware depends on the standard. In this section, only implementation independent details common to all uplink implementations are described. Implementation dependent details of the interaction, including sub-protocols, are given for the USB uplink implementation in Section 4.1, which clarify what type of requirements must be achieved for a whole uplink communication implementation.

It is also important to note that the nature of the uplink communication is generally asynchronous, since the CPU initiates transactions at random times, and responses to request transactions are also sent by the firmware in an asynchronous manner. However, synchronous implementations are also possible, since software components on the CPU may buffer requests and send them to the bridge periodically. For example, a USB implementation has to be synchronous since USB uses a periodical polling mechanism for communication. However, an RS232 implementation must be asynchronous since peers send all data serially without explicit synchronization.

3.3.1.1 Uplink Transactions

Uplink communications are carried out by means of transactions between the CPU driver and the bridge uplink interface. These transactions are categorized under two types, *requests* and *responses*, as seen in Figure 3.5. Requests are transactions initiated from the the CPU side, targeted at the bridge, whereas responses are transactions from the bridge to the CPU consisting of the replies to requests. Request transactions can be either *bridge requests* or *node requests*, and the same classification applies to responses, which can be either be *bridge responses*, or *node responses*. The distinction between bridge and node transactions is important, since the interpretation of the two types at the bridge firmware is different. Bridge transactions encode data and command exchanges between the CPU and the bridge, and are not visible to users, since the bridge is not visible in the logical topology. Therefore, bridge requests are only initiated internally

by CPU libraries, to control the bridge. On the other hand, node requests can be initiated by both CPU libraries and user applications, and they are forwarded to nodes by the bridge receiving the request. A user application that controls a distributed system via the URB processes all data and command exchange with nodes using node requests and responses.

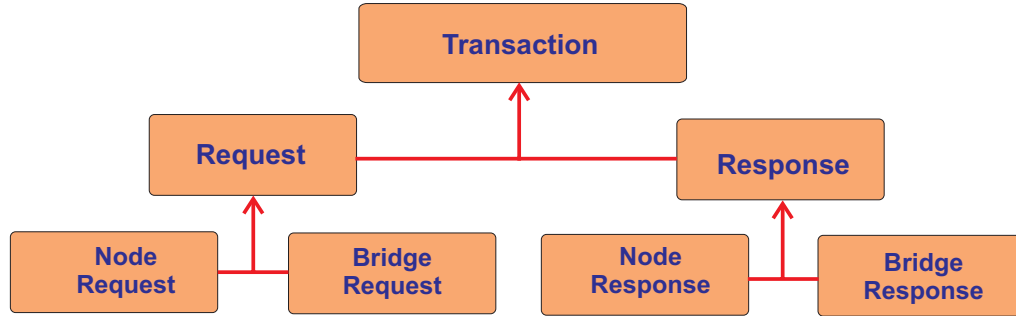


Figure 3.5: URB Transaction types.

Uplink transactions require exchange of uplink packets between the CPU driver and the bridge uplink interface. Categorization of packets is shown in Figure 3.6, where *Job Dispatch Packets (JDP)* refers to request transactions, and *Job Response Packets (JRP)* refers to response transactions. A JDP can be one of two types: a *Normal Job Dispatch Packet (NJDP)* or an *Urgent Job Dispatch Packet (UJDP)*. Any request transaction, whether targeted at the bridge or the node, can be realized by the transmission of a NJDP, or an UJDP in case of an urgent transaction. The categorization of transactions and packets is framework specific, and throughout the rest document, the phrase *requests* is used to denote request transactions using JDP, and *responses* is used to denote response transactions using JRP. It is also acceptable to use the term *urgent request packet* for a UJDP, and *normal request packet* for a NJDP.

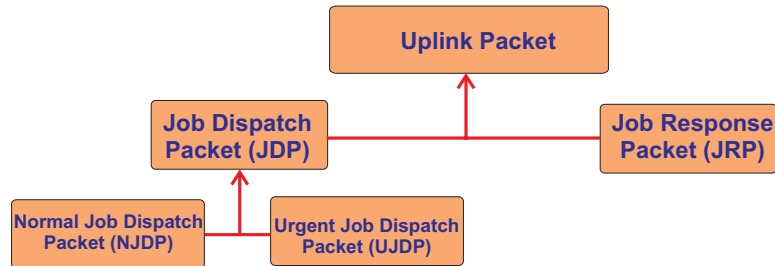


Figure 3.6: Packet types for the URB Uplink.

3.3.1.2 Packet Formats

The generic format of a URB uplink packet is shown in Figure 3.7 and consists of the following items:

- The first byte consists of the following fields:
 - **Size-2**: Five least significant bits encode a value which is two less than the packet size. As a consequence of this scheme, the maximum size of a packet that can be obtained using this generic format is 33 bytes.
 - **Flags**: Remaining three bits encode some flags whose details are given below.
- The second byte consists of one of the following:
 - **Opcode/Content** (bridge transactions): This byte encodes the 8-bit opcode of the bridge request, whose details can be found in Section 3.3.1.3, for requests and is the first byte of the response content for response transactions
 - **Address** (node transactions): For both requests and responses this byte encodes the destination address consisting of the 4-bit URB node address, the 3-bit message box ID, and a bit to decide whether the request is a read or write. As result of this encoding, the URB supports up to 16 nodes per bridge, as well as 8 inboxes and 8 outboxes on each node.
- The rest of the packet consists of the packet content, whose details can vary according to the type of packet.
 - The content of a bridge request is simply arguments of the command, whereas the content of a bridge response starts from the second byte.
 - The content of a node transaction depends both the type(request/response) and the direction (read/write). Details can be found in Appendix B.2.

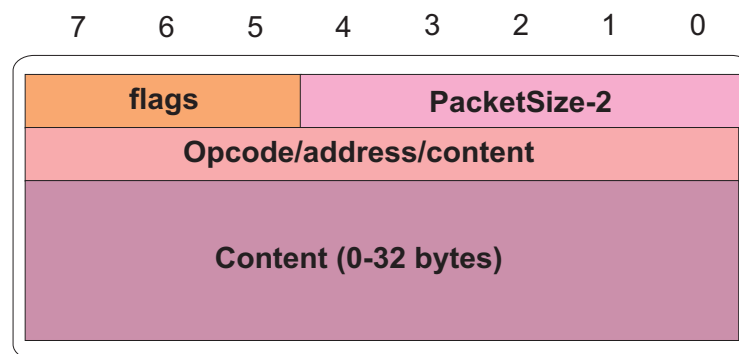


Figure 3.7: The URB Generic Uplink Packet Layout.

The flags field of an uplink packet may consist of a collection of the following flags:

- **BR(7):** The BRidge flag is common to all types of packets and if it has the value of 0, the packet is a node request, otherwise the packet is a bridge request.
- **RR(6):** The Response Required flag is only available in requests and indicates whether a response is required for this request, having the value of 0 if a response is not required, otherwise it's equal to 1.
- **ER(6):** The ERRor flag only exists in bridge responses, specifying an error occurred when set to 1. If it's set due to an error, the response content includes a single byte encoding an error code, instead of the actual response.
- **AR(6):** The Automatic Response flag, which exists in node responses, indicates whether the request was generated as a result of an explicit request, where it's set to 0, or automatically by the bridge, where it's set to 1. Autonomous data acquisition is explained in Section 3.2.2.
- **UR(5):** The URgent flag is common to all types of packets and has the value of 1 for urgent requests, otherwise it's equal to 0.

3.3.1.3 Bridge Commands

The bridge is controlled by the CPU using the bridge commands whose full description can be found in Appendix B.1.

3.3.2 Downlink Communication

Downlink connectivity is currently designed and implemented for the I^2C standard due to its compatibility with a broad range of dedicated microcontrollers. The design of the URB with the bridge as a gateway allows new and high performance communication protocols on the CPU side to communicate with old and cheap microcontrollers through the I^2C protocol. Although the current implementation of the URB framework only supports I^2C as the downlink, other standards mentioned in Section 1.3 can also be integrated into the system by implementing the required downlink software.

Transactions between the bridge and its associated nodes adopt an abstraction in the form of *message boxes*, which can be an inbox for receiving input, or an outbox for sending output. Each node is allowed to have up to eight inboxes and eight outboxes, each with a static fixed size between 1 and 33 bytes selected during initialization. The bridge, being the master of the downlink connection, can request data to be read from or written to any of these message boxes based on requests coming from its uplink.

Each transaction encodes, either explicitly or implicitly, a URB message box address, consisting of a 4-bit node address, a 3-bit message box id and a read/write flag. Sufficient information is provided in this address to identify unambiguously a single message box on a single URB node. Outbox 0 and Inbox 0 are reserved for providing node configuration information and sending URB related commands to a node, respectively. The size and layout of these message boxes are specified by the URB protocol and implemented by all relevant libraries, and further detail on their layout can be found in Appendix A.1.

Although I^2C is a multi-master protocol, URB is designed such that the bridge is the only master of the downlink and all the nodes connected to it are slaves. Therefore, downlink communications can only be initiated by the bridge, and the bridge processes downlink transactions synchronously by polling the bus every 1ms. Downlink communications have a considerable amount of other details, including types of downlink transactions and commands, node synchronization scheme, which can be found in [4], and Appendix A.1. Nevertheless, information given in this section will be sufficient for the rest of the document.

3.4 URB APIs

In this section, APIs provided to the users are described, including details of the URB CPU API, consisting of the calls used by the user application on the CPU, and the URB Node API for the local node firmware. This thesis has partial contributions to the CPU API, and no direct contributions to the Node API.

3.4.1 URB CPU API

A URB system can employ multiple buses running in parallel, controlled by separate bridges. The CPU API abstracts away from these details and presents a uniform collection of devices to the programmer. In this section, the CPU API is briefly explained and some of its important properties are highlighted. An overall view of the CPU API, including the relationships between the classes, is illustrated in Figure 3.8.

At the core of the CPU API, the `URBInterface` class is defined as a singleton [13], which provides a unified interface to URB functionality for all buses connected to the computer. Derived instances of the `BusManager` class for each type of uplink connection register to this interface and handle connectivity to low level bridge device drivers. These bridge drivers, as well as derived instances of `BusManager` class, are specific to the uplink implementation, and further details

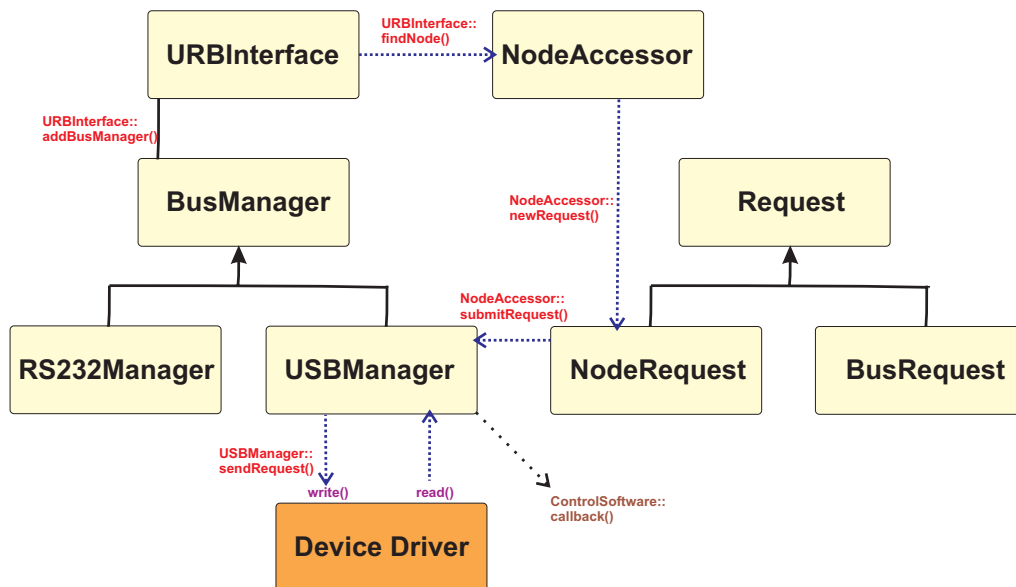


Figure 3.8: The CPU API.

are available on Section 4.4, where the USB instantiation of the API is described.

The **BusManager** class encapsulates a common abstract interface to all URB uplink implementations, where new uplink connections are established by deriving a new class from this one and implementing the required virtual methods. This base class also provides facilities to store and query information about nodes attached to this manager as well as an efficient allocation pool for **BridgeRequest** instances. It is also important to note that a derived **BusManager** instances may be managing multiple bridges of the same type. For example, **USBManager** is a derived child of **BusManager**, and bridges connected to the CPU can be managed by an instance of **USBManager** class per bridge.

The **URBIInterface** class supports search queries for nodes connected to the system through its `findNode()` method, based on class and index specifications of the nodes, which is described in Appendix A.1. This method returns a **NodeAccessor** pointer, where a **NodeAccessor** object encapsulates all information necessary to access a particular node. The **BusManager** instances create a single **NodeAccessor** object for every node they discover on their managed buses. Once a node has been located, its associated **NodeAccessor** object provides methods for creation and submission of **NodeRequest** objects, transparently handling

all buffering issues and other details of uplink communications. **NodeRequest** objects are derived instances of the base class **Request**, which handles basic request functionality, including callback management, request status as well as management of data packets. **BridgeRequest** objects are also derived from this base class, as mentioned in Section 3.3.1.1, but since bridges are hidden from the user, these objects are never created by the users of the CPU API.

The URB CPU API is designed and implemented in the C++ language. This summary provides a general overview of the API, and further details can be found in the API documentation.

3.4.2 URB Node API

The URB framework provides a simple Node API to facilitate rapid development of application-specific firmware on URB nodes. The current implementation of the URB Node API and its relevant libraries support 8051-compatible microcontrollers of Silicon Laboratories (Silabs). Most of the functions declared by the Node API are implemented by the node libraries, which handle local configuration of nodes and downlink communication details. Implementation of some functions of the Node API which involve firmware dependent activities are left to the user. All the details of the URB Node API can be found in [4].

Chapter 4

USB Uplink for URB

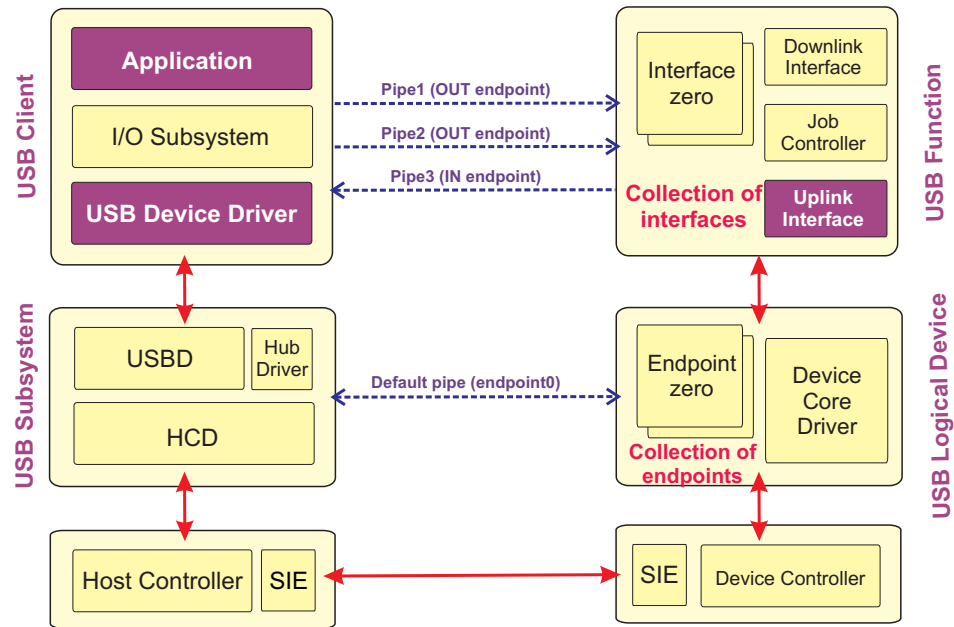


Figure 4.1: A layered component decomposition of the USB uplink for the URB system.

The Universal Serial Bus uplink is designed to provide improved uplink connectivity to the URB architecture, while benefiting from the advantages of the well known and widely used USB protocol. In Figure 4.1, the layout of the USB uplink for URB is shown, including software and hardware layers of the host (CPU) and the bridge components. As mentioned in Section 3.3.1, the uplink is the connection between the host, which may be an IBM compatible PC for

experimental purposes, or a PC104 card for a mobile robot, and the bridge, a dedicated microcontroller with USB support. This connection is physically established by connecting the USB ports of the two components through a single USB cable.

Although the uplink communication consists of several layers, the scope of the USB uplink implementation concerns only the shaded layers of Figure 4.1, which are the control software, the USB device driver, and the bridge uplink interface. The remaining layers are low level details, and are partially described in Section 2.4.

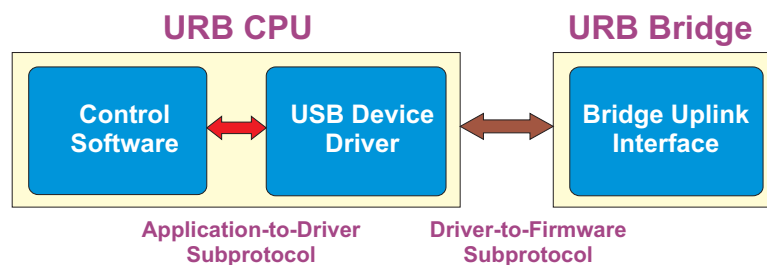


Figure 4.2: Simplified software layout for the USB Uplink of URB.

A simplified view of the uplink system can be seen in Figure 4.2, where only components that are implemented under the scope of the USB uplink are shown. Details of the communication protocols are given in Section 4.1, taking into account both the Application-to-Driver subprotocol and the Driver-to-Firmware subprotocol. Then, the bridge uplink interface is briefly explained in Section 4.2. Next, the driver for Linux environment is explained in Section 4.3, comprising the details on file operations, buffering decisions, concurrency management issues, and the algorithm. Afterwards, the control software is presented in Section 4.4, where the details of the USB instantiation of the URB CPU API, shortly referred to as *USB libraries*, are given in Section 4.4.1. Finally, the client of the CPU API, the user application, is explained in Section 4.4.2. One final remark is that the control software is an executable that consists of the user application and the static USB libraries linked to each other. Throughout the rest of the document, the terms *Application* and *Control Software* are used interchangeably to mean the same user level process.

4.1 Communication Decisions

Since the USB host communicates with the peripherals using a polling mechanism as mentioned in Section 2.4, the USB uplink also communicates with bridges synchronously, as opposed to other protocols such as RS232. The USB uplink connectivity consists of Application-to-Driver and Driver-to-Firmware communication subprotocols, which are compatible with the generic URB uplink communication protocol. In this section, the usage of the generic uplink protocol for this USB implementation is briefly explained.

4.1.1 Application-to-Driver Subprotocol

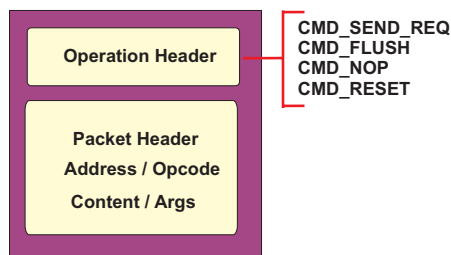


Figure 4.3: General request packet format for the Application-to-Driver Subprotocol.

The subprotocol between the application and the driver uses packets that are formed by inserting a one-byte operation header to the generic uplink packet as shown in Figure 4.3. There are four values that this operation header can take: `CMD_NOP` (0xA3), `CMD_RESET_DEV` (0xFD), `CMD_SEND_REQ` (0xFE) and `CMD_FLUSH` (0xFF). There is an agreement between the application and the driver on what these operations are, and how they are handled by the device driver. The header is only meaningful between the application and the driver, and the rest of the system has no knowledge about this subprotocol. The meanings of the uplink headers are as the following:

- `CMD_SEND_REQ`: The following packet is to be buffered by the driver
- `CMD_FLUSH`: Initiate transmission of buffered packets

- CMD_NOP: No operation
- CMD_RESET: Reset communication with the bridge

There is also traffic in the reverse direction, which involves the retrieval of response packets by the application. This reverse traffic does not include operation headers, and consequently, transmitted packets are the same as the generic uplink packets.

4.1.2 Driver-to-Firmware Subprotocol

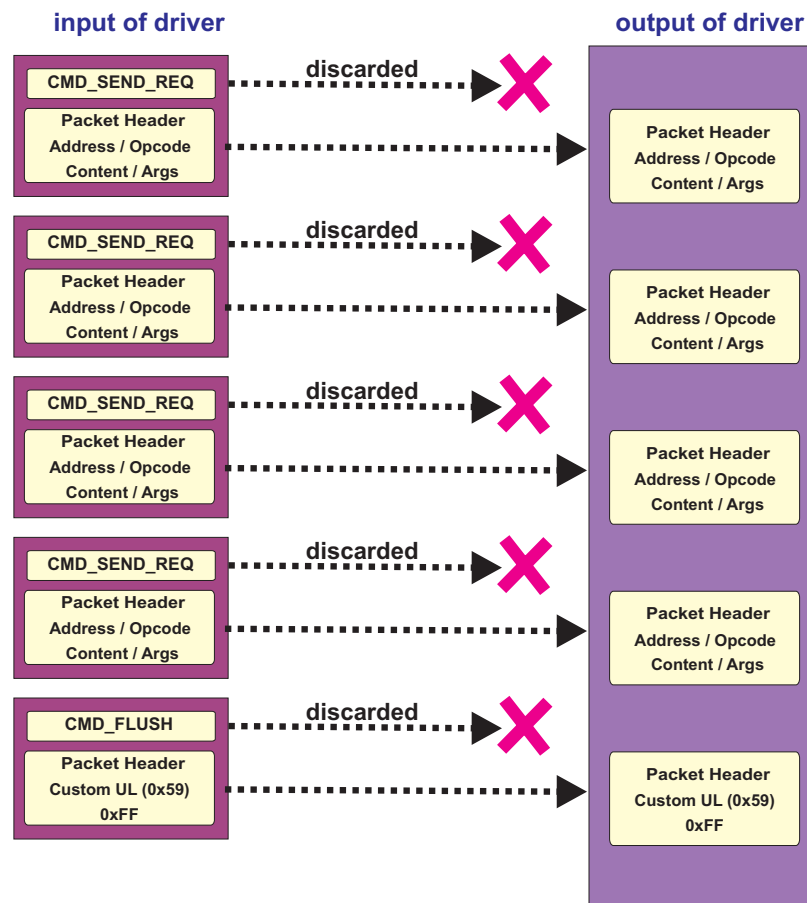


Figure 4.4: The processing of a packet by the driver: incoming packet is interpreted by the driver according to the operation header, operation header is discarded, and remaining payload is buffered.

Upon receiving packets sent by the application, the driver decodes them and extracts operation headers. The driver then processes packets according to their operation headers. In particular, if the operation header is a `CMD_SEND_REQ`, the driver buffers the packet into a kernel buffer specific for writing to the device, i.e. the *write buffer*. However, if the operation header is a `CMD_FLUSH`, the driver adds an uplink custom bridge command for flush operation to the kernel buffer, illustrated in Figure 4.4, and initiates transmission of the entire kernel buffer to the bridge uplink interface (Refer to Appendix B.1 for bridge commands and Appendix B.2 for custom command packet format). The header can also be a `CMD_NOP`, which is used for synchronization purposes, where the driver simply discards the packet. Finally, the driver can receive a packet with the header `CMD_RESET`, which initiates a bus reset to the firmware, and is used when the synchronization between the CPU and the bridge is lost. Decoding and interpretation of the operation headers are processed in a state machine within the driver. Further details about the state machine can be found in Section 4.3.1.

The subprotocol between the driver and the firmware is the same as the generic Uplink Communication, where entire buffered packets are transmitted within a USB Request Block (urb). Therefore, the subprotocol between the driver and the firmware is a copy of the generic uplink protocol including transmissions in both directions, and no extra overhead is introduced. It is important to note that these packets undergo many modifications in downward layers of the USB subsystem, such that they are firstly converted to transactions by the addition of USB protocol wrappers, as explained in Section 2.4. The USB formatted packets are finally converted to electrical signals by the HC, where signals propagate through the USB port and depart the host. These low-level physical details are not in the scope of this thesis, and further details regarding low-level details of USB can be found in [8].

4.1.3 The Uplink Transfer Policy

Using a Control Transfer at probe time is inevitable for the configuration of the USB device, but other transfer types are used for data exchange once a device

is successfully configured and is ready to use. Therefore, determining the type of USB transfer to use for data exchange that best meets the communication requirements of the device and the application is an important issue.

Table 2.1 provides a useful comparison of USB transfer types to help determine the appropriate uplink transfer policy. Since the fastest mode supported by the Silabs F340 board is the full speed mode, we are only interested in the transfer characteristics of this mode. Different aspects of the USB module within the F340 board, summarized in Section 2.6, are also important in determining the proper choice of transfer policy.

Since one of our primary constraints in URB is reliability, we cannot rely on the isochronous transfer type because it does not retry packet delivery in case of errors. Aspects of the remaining two types, bulk and interrupt modes, are the same in terms of the maximum payload size and protocol overhead. Although the interrupt mode has advantages over the bulk mode in terms of bandwidth priority and latency guarantee, several factors make the bulk mode more preferable for our communication needs.

First of all, the bursty data transfer nature of bulk mode is more appropriate, since the URB CPU does not expect a data stream at a constant rate. Secondly, the bulk mode is more suitable for data transfer in large amounts, so that the HCs usually assign bulk transfer a larger maximum payload size (remember that the maximum payload sizes in Table 2.1 are only upper limits, the HC is free to determine a lower payload size depending on the current settings of the bus), and give priority in transmitting the split transactions belonging to a specific bulk transfer in the same or consecutive USB frames. It is important to note that interrupt transfers have at most one transaction per frame belonging to a specific data transfer, which makes it inappropriate to transfer large amounts of data [14]. Finally, the bus we intend to use for the USB uplink is not likely to have any other USB devices, and all the available bandwidth is likely to be used by the bulk transfer so that no problems with the latency are expected. Taking into account these reasons, we chose the Bulk mode as our USB uplink transfer policy.

4.2 Bridge Firmware

The Serial Interface Engine (SIE) sub-component of the USB Function Module, depicted in Section 2.6, is in charge of all low level protocol tasks such as transformation from USB formatted packets to normal packets, interruption of the microprocessor upon transmission or reception of data, and generation of handshake signals. In our design, the bridge firmware is responsible for providing a USB Logical Device abstraction, as well as satisfying other functional requirements of the client layer, all illustrated in Figure 4.1.

The bridge firmware incorporates a top level ISR within device core driver of USB Logical Device. Upon the generation of interrupts by the underlying hardware, this top level ISR calls the appropriate interrupt handlers. The default pipe, Endpoint0, is configured by the driver in a similar fashion, where SIE generates interrupts upon reception or transmission of control packets on Endpoint0 FIFO, and required utility functions for the configuration are called by the driver.

The Client layer of the bridge firmware is expected to satisfy all the functionality mentioned in Section 3.2.2, including mastering of the downlink, synchronization across nodes, and scheduling of automatic requests, as well as providing connectivity between the uplink and downlink channels. All of these functionalities are implemented by the job controller layer, and details of these functionalities, which can be found in [4], are not directly related to the USB uplink implementation.

The Uplink Interface is our major focus within the overall bridge firmware structure, as shown in Figure 4.2, since it is responsible for the communication of the bridge job controller with the USB Host Client through the pipes between the two. There are three stream pipes used by the Uplink Interface, and one message pipe used by the USB Logical Device, all of which are summarized in Table 4.1 and shown in Figure 4.1. In Table 4.1, direction is indicated from the perspective of the host, so that pipes from the host to the bridge are OUT, and from the bridge to host are labeled IN. Split mode is not required since the number of pipes is sufficient for our needs.

Endpoint	Direction	Double Buffering	Max FIFO Size	Function
Endpoint0	IN/OUT	-	64	configuration
Endpoint1	OUT	enabled	64	urgent re-requests
Endpoint2	OUT	enabled	128	normal requests
Endpoint3	IN	enabled	256	responses

Table 4.1: Endpoint Configurations of the Bridge Firmware. Direction is from the perspective of the CPU.

There are two buffers on the bridge: hardware FIFOs for accumulating incoming packets, and software ring buffers which accept packets forwarded by the FIFOs. The sizes of the FIFOs, which determine the maximum size of data that can be buffered within a USB frame, are constrained by the board hardware. As shown in Table 4.1, their sizes are halved when compared with Figure 2.8 as a result of enabling double buffering, which is preferred so as to provide a more reliable system by ensuring that data in buffers are never overwritten. Sizes of ring buffers are carefully determined by the firmware implementer, as 512 and 1024 bytes for *receive_buffer* and *send_buffers* respectively, since available RAM on the microcontroller is limited to 4352 bytes.

An important constraint on the firmware arises from the usage of the bulk transfer mode, which limits the maximum size of a packet to 64 bytes. Therefore, although the maximum sizes of the endpoint FIFOs are higher than 64 bytes, the HC splits the data it receives from the kernel in the form of a single urb into 64-byte transactions, and sends them one after another until all data within the urb is transmitted to the OUT FIFOs on the bridge. Upon receiving a transaction, the top level ISR is invoked to read data on the FIFO into the *receive_buffer*, where data is processed to produce responses. The communication bandwidth with the bridge is limited by the speed of the firmware to process the data on *receive_buffer*, and forward the responses to the *send_buffer*. Responses accumulated on the *send_buffer* are split into 64-byte packets by the firmware, undertaking the functionality of HC on the device side, and sent through the IN FIFO as 64-byte transactions.

One final remark is that the term *firmware* is used throughout this document to refer to the Uplink Interface, since the uplink implementation is the primary focus of this thesis.

4.3 The Linux USB Device Driver for URB

We have implemented the USB device driver as a Linux kernel module. It consists of core components composed of all file operations as well as functions needed for driver and device installation, and two helper subprograms for buffer management and time measurement. This module implements a character oriented USB device driver for interfacing to the Silabs F340 board.

The core of the USB driver consists of several structures and functions, described in Section 4.3.1. It is convenient to give details about the most important structure:

```
struct urb_usb_t {
    struct usb_device *udev;
    struct usb_interface *interface;
    struct kref kref;
    bool attached;
    __u8
        bulk_in_endpointAddr,
        urg_bulk_out_endpointAddr,
        bulk_out_endpointAddr;
    size_t
        bulk_in_endpointSize,
        urg_bulk_out_endpointSize,
        bulk_out_endpointSize;
    struct usb_fifo *write_buf, *urg_write_buf, *read_buf;
    wait_queue_head_t writeq, readq, bufq;
    FLAG_T    read_complete_f, data_ready_f, buf_full_f;
```

```

    BYTE      *temp_read_buf, * temp_write_buf;
    struct urb *r_urb, *w_urb;
    BYTE      *r_urb_dma_buf, *w_urb_dma_buf;
    FLAG_T    rcb_done_f, wcb_done_f;
    spinlock_t buf_lock, data_ready_lock, w_lock, dev_buf_lock;
    phase_t    w_state;
    UINT      bytes_to_buffer;
    atomic_t   dev_available, write_allowed;
    int        _cReset;
    BYTE      _cWait;
    int        write_counter;
}

```

The structure `struct urb_usb_t` is a global structure to hold device specific information, and is accessed by all functions in order to manipulate device properties. Most commonly used fields of the structure are the buffers, `write_buf` and `read_buf`, which are used for buffering data that will be sent to the device, and received from the device, respectively. These buffers are structures of `struct usb_fifo`, which is defined in the buffer management subprogram for which further details are available in Section 4.3.2. Device descriptor and interface descriptors of the `struct urb_usb_t` are also used throughout the device driver for accessing the internal properties of the device and the interface. There are three wait queues in `struct urb_usb_t`, on which the calling routines wait for a specific event. Various flags belonging to `struct urb_usb_t` are used for synchronous I/O operations such that calling routines sleep or wakeup according to values of these flags. Spinlocks of the structure are used for resolving shared data problems, and protect data which are accessed from multiple threads. Lastly, `phase_t` field is used for holding the current state of the state machine, used in write function, and illustrated in Figure 4.5.

4.3.1 Driver Functionality

The sequence of events that a USB driver needs to follow for registration are given in Sections 2.3.1 and 2.5. The functions implemented for driver registration and unregistration are as follows¹:

- `int urb_usb_init()` : This function is called for registering the USB driver to the USB Core. The USB driver is assigned a specific Vendor ID and a Product ID corresponding to the device it is charged to handle.
- `void urb_usb_exit()` : This function is called for unregistering the USB driver from the USB Core.

The functions for device registration and unregistration are as follows:

- `int urb_usb_probe(struct usb_interface *, struct usb_device_id *)`:
This function is called by the USB Core when a USB device with the Vendor/Product ID pair that matches the values introduced in the driver registration process is plugged to a USB port of the host. The routine uses the parameter `struct usb_device_id` to extract the Vendor/Product IDs, and the device is defined by the corresponding interface, `struct usb_interface`. The probe function firstly initializes the variables of the `struct urb_usb_t` structure, including the setting up of the endpoint addresses. Afterwards, it registers the device defined by `struct urb_usb_t`, whose fields are initialized with the appropriate values, to the USB Core by obtaining a minor number for this device.
- `void urb_usb_disconnect(struct usb_interface *)` : This function is called by the USB Core either when the device defined by `struct usb_interface` is removed from the host, or when the driver is unloaded from the USB Core.

The file operations of our USB driver are as follows:

- `int urb_usb_open(struct inode *, struct file *)`: This function is called by the I/O subsystem when it receives an `open` call from the application layer.

¹Any of given function signatures can be changed with future implementations of kernel usb modules. The current implementation of functions is realized with kernel version 2.6.29.

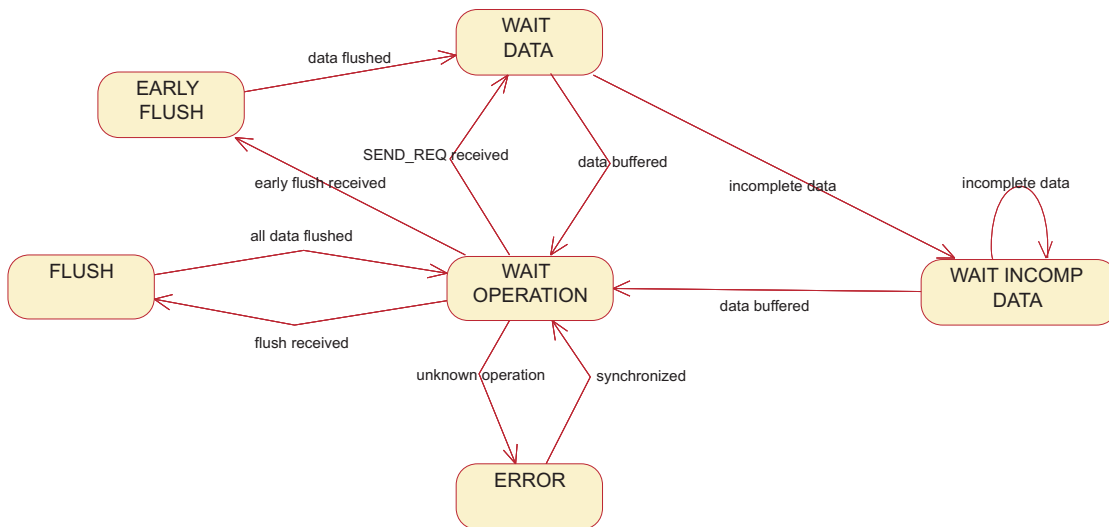


Figure 4.5: State Machine of the write function.

The routine first extracts the minor number from the `struct inode` and retrieves the `struct urb_usb_t` corresponding to the minor number. Afterwards, it tests whether the device is already open or not, and if not, the kernel file descriptor, `struct file`, is assigned the pointer of `struct urb_usb_t` for easy access in read and write routines. It is important to note that a bridge can be accessed by only one application at a time in our design, so that any application that tries to open an already in use device will be put to sleep until the application that currently uses the device finally releases it.

- `int urb_usb_release(struct inode *, struct file *)`: This function is called by I/O subsystem when it receives a `close` call from the application layer, and releases any resources the `open` routine has allocated.
- `ssize_t urb_usb_write(struct file *, char *user_buf, size_t count, loff_t *)`: This function is called by I/O subsystem when it receives a `write` call from the application layer. This routine firstly copies `count` bytes of data from the user space to a temporary kernel buffer, and then processes the retrieved data according to the current state, `phase_t`, of `struct urb_usb_t`. The states, illustrated in Figure 4.5, can be one of the following:
 - `WAIT_OPER`: The operation header is extracted and the new state is determined according to extracted header.

- **WAIT_DATA**: The content of the packet is extracted and buffered into the **write_buf**. If the entire packet is not available in the temporary buffer, the state is set to **WAIT_INCOMP_DATA**.
 - **WAIT_INCOMP_DATA**: The rest of the incomplete data packet is buffered into the **write_buf**.
 - **FLUSHING**: The flushing state initiates a sequence of events, which consists of the transmission of buffered request packets to the device, and the reception of response packets. The flushing concept can be understood better after reading Section 4.3.4, where the full algorithm is given.
 - **EARLY_FLUSH**: An early flush is initiated by the driver when a flushing operation is required to prevent buffer overflows. It is important to point out that although a normal flush operation is requested by the upper level application, an early flush is only decided and initiated by the driver. There is no difference between a normal and an early a flush in terms of the operations that follow after.
 - **ERR**: Recovery from an error condition is achieved by breaking out of state machine, and returning with an error code.
- **ssize_t urb_usb_read(struct file *, char *user_buf, size_t count, loff_t *)**: This function is called by the I/O subsystem when it receives a **read** call from the application layer. This routine does not communicate with the device due to our design of the driver. It simply sends requested number of bytes from the **read_buf** to the application, returning the number of bytes that were successfully sent.

Completion handlers, which are depicted in Section 2.5, are as follows:

- **void urb_usb_wcb(struct urb *w_urb)**: This handler is called by the USB Core upon completion of the write urb submission process, either successfully, or with an error. This write callback function firstly checks the status of the transmission, and returns in case of a failed write urb submission. If the submission is successful, a urb for retrieving the response packets from the device, which can be referred to as *read urb(r_urb)*, is initialized, and submitted to the USB Core.

- `void urb_usb_rcb (struct urb *r_urb)`: This handler is called by the USB Core upon completion of the read urb submission process, either successfully, or with an error. Upon a successful reception of responses, this read callback routine adds the response data to `read_buf`, and wakes up `write` and `read` threads if they are in sleep.

4.3.2 Buffering Decisions

Buffer management is necessary for transferring data between the kernel and a device in an efficient manner. Our buffer management subsystem is implemented for this purpose, where allocation, initialization and clean up of buffer space, as well as insertion and retrieval of data, and other utilities are handled.

In the main program, two kernel buffers are allocated on the heap: `write_buf` for buffering data that will be sent to the device, and `read_buf` for buffering data received from the device. While `read_buf` is required to avoid losing data when there is no application reading at the instant of response arrival, `write_buf` is used for efficiency reasons.

Since a pipe is a stream between a kernel buffer and an endpoint, the size of a kernel buffer and the corresponding device FIFO should be closely related with each other. However, since the host controller sends 64-byte long bulk transactions to the device endpoint FIFO at full speed, and data at the FIFO is read by the firmware into ring buffers immediately, establishing a relationship between sizes of kernel buffers and firmware ring buffers is a more convenient approach. Consequently, the sizes of kernel buffers are determined according to maximum sizes of ring buffers on the device side. It is important to note that a small buffer size might decrease performance by frequent buffer overflows, whereas a large buffer size results in a waste of kernel resources. Taking these into account, we follow a generally accepted convention [8] and choose the sizes of buffers to be four times the size of the corresponding ring buffers: `write_buf` is 2048 bytes long and `read_buf` is 4096 bytes long. The reason for choosing buffer lengths larger than the firmware ring buffer sizes is to provide efficiency, since it is possible to

be buffering data while transmitting a maximum sized data block. Note that the maximum size of a data packet is constrained by the maximum FIFO size, and an attempt to transmit a packet with a payload larger than the FIFO size will fail, returning a babble error code. However, since the host controller guarantees transmission of 64 byte transactions at a time, the situation of babble error is not possible with our design.

It is also important to note that buffer manager sub-program implements a *ring buffer*, which is a fixed size data buffer that uses two pointers, i.e. head and tail, for tracking the elements removed from or added to the buffer. The primary advantage of a ring buffer is that its elements do not need to be shuffled around the buffer in case of data removal, so that it is appropriate and efficient for data streaming.

The reason for using a buffer instead of immediately sending any incoming packet from the application is that the efficiency of communication increases when the number of bytes per urb is closer to the maximum value the pipe can support, and the minimum number of transactions possible are used for the transmission of a certain amount of data. The full speed mode of F340 board uses a USB framing mechanism with 1ms intervals, and usually only a limited number of transactions corresponding to a urb can be transmitted through a pipe within a frame. Therefore, a buffer which stores incoming data for a specific time interval less than or equal to 1ms, and flushes all the buffered data within a single urb contributes much to the overall communication performance in terms of bandwidth. This hypothesis is also supported by tables in the USB Specification [8] where it can be observed that while a bulk transfer in full speed mode with a data payload size of 1 byte transmits 107 bytes of data using 107 transactions within a frame, a transfer that uses a 64 bytes of payload transmits 1216 bytes of data using only 19 transactions. The scenario shows the statistics for the best case, and since there is no guarantee that all of the 107 transactions will be scheduled within a single frame, they may certainly be sent in more than one frames by the USB Host Controller, which will add more latency accordingly. All of this evidence shows that our buffering mechanism with the concept of flushing, contributes to the communication performance of the URB.

The buffering concept is also taken into account for the application layer. There is a temporary read buffer in the application for receiving data from the `read_buf` of the kernel. In contrary, there is no write buffer at the application because incoming requests are appended into a linked list. The interaction between application read buffer, linked list and driver buffers is explained with more details in Section 4.4.1.

4.3.3 Concurrency Management Issues

Concurrency management is one of the most important concerns of kernel programming in order to provide robust software. Sources of concurrency, as well as several strategies to resolve concurrency problems and Linux concurrency primitives are explained in Section 2.3.2. During design and development of the driver, two strategies were followed: correct sequencing of interactions in main program flow, and coordination of access to shared resources while minimizing the amount of shared data.

The general program flow is described in Section 4.3.4, where we can see that the flow is divided into two threads of execution according to current implementation: a write thread and a read thread. The write thread handles all communication with the device, including both transmission and reception of data, and consists of the functions `write`, `wcb`, and `rcb`. These three functions never run concurrently, and always run in the same order, such that at first `write` runs, and it is put to sleep upon sending a write urb to the device; then `wcb` runs and terminates after sending a read urb to the device, and finally `rcb` runs waking up the sleeping `write` thread. The read thread on the other hand, consists of only `read` function, and simply sends data in the `read_buf` of the driver to the application. The program flow is controlled by the Blocking I/O mechanisms, i.e. process sleeping and waking up functions of Linux kernel, which are explained in Section 2.3.3, and various flags belonging to `struct urb_usb_t`.

Since there are two threads of execution, and it is not necessary to take precautions against race conditions within a thread, the coordination of access to shared

resources is not very complex for this driver. Global data used in both threads, which are mostly fields of the structure `struct urb_usb_t`, i.e. the flags, are protected by spinlocks. The reason for preferring the spinlock mechanism is that the `wcb` and `rcb` functions of the write thread run in interrupt context, and in contrast to other primitives, spinlocks can reliably be used in such situations. Moreover, since code segments embedded between the locking and unlocking statements of spinlock implementation are generally short throughout the driver, threads are likely to be blocked for short time periods, which makes the spinlock superior over other primitives such as semaphores, due to prevention from frequent context switches.

Buffers are data structures that are most exposed to race conditions, since `read_buf` and `write_buf` are frequently accessed in both threads, and it takes more CPU cycles to process a buffer operation. Consequently, concurrency issues are automatically handled in the buffer management sub-program, abstracting away the details from the users of the buffer. For instance, when the main program needs to retrieve `n` bytes from a buffer, it simply calls the corresponding routine of the buffer without dealing with concurrency.

When we have a spinlock that can be taken by the code that runs in an interrupt context, we must use a spinlock that disables interrupts in order to prevent a deadlock. There are variants of spinlocks to be used in such special situations, and in our driver we used `spin_lock_irqsave()` from the library `linux/spinlock.h`, which disables interrupts on the local processor and stores the previous interrupt state in a local variable. Afterwards, we used `spin_unlock_irqrestore()` to unlock and restore the previous interrupt state.

4.3.4 Algorithm

The USB Driver module, `urb_usb`, consists of several subprograms, and it is not feasible to give all the algorithms here. Instead, it is more suitable to explain the core algorithm, which uses the flushing concept and two threads of execution for establishing the communication between the device and the application, using

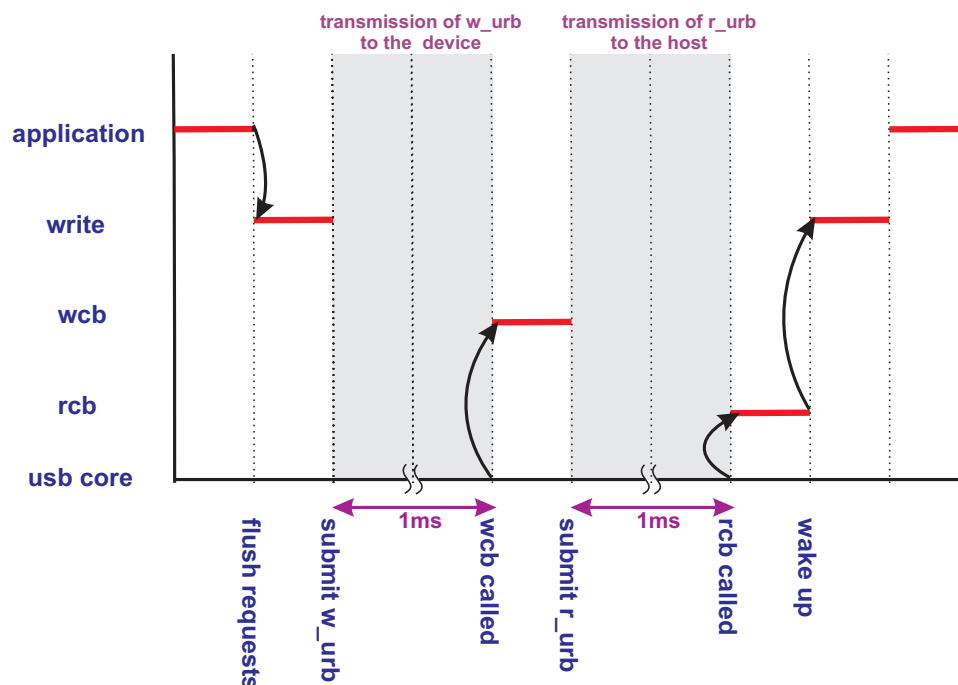


Figure 4.6: Timeline of events initiated by a call to the write function within the write thread.

the protocols depicted in Section 4.1.

Before we describe the details of the algorithm, it will be more convenient to explain the flow of events. As mentioned in Section 4.3.3, there are two threads of execution involved in the communication of the driver with the device, and with the application. The read thread is straightforward, since it consists of only one procedure, and does not directly interact with the device. On the other hand, the write thread handles all communications, with the three routines it contains. The timeline of events in write thread is illustrated in Figure 4.6, where we see that there is a loop of write-wcb-read-write events for data transfers. Note that the figure is not drawn to scale, since execution times of instructions at the kernel are very fast relative to the 1ms polling period of the USB Host Controller.

Activity diagrams illustrated in Figures 4.7, 4.8, 4.9 and B.2 provide a more clear understanding of the algorithm. While the flow of events in `write` function is displayed in Figure 4.7, Figure 4.8 shows the sequence of activities in callback functions, all of which belong to the write thread of execution. The activity

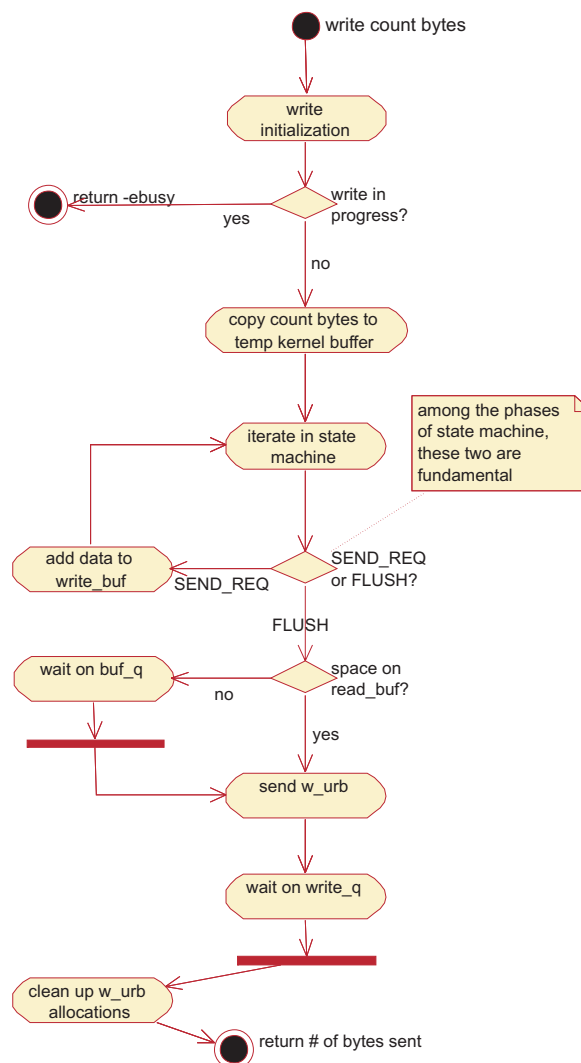


Figure 4.7: Activity diagram showing the flow of events involved in the write function.

diagram in Figure 4.9 illustrates the flow of events in the read thread. These figures illustrate algorithms for individual functions, without interactions between the routines such as signaling of a running thread to wake up a blocked thread. In these figures, blocked (sleeping) state of the threads is shown by means of a horizontal bar. Lastly, both of read and write threads including the interactions are shown in Appendix B.4, due to space limitations.

It is worth describing the write function of the write thread in more detail. The state machine given in Figure 4.5 shows the transition between different

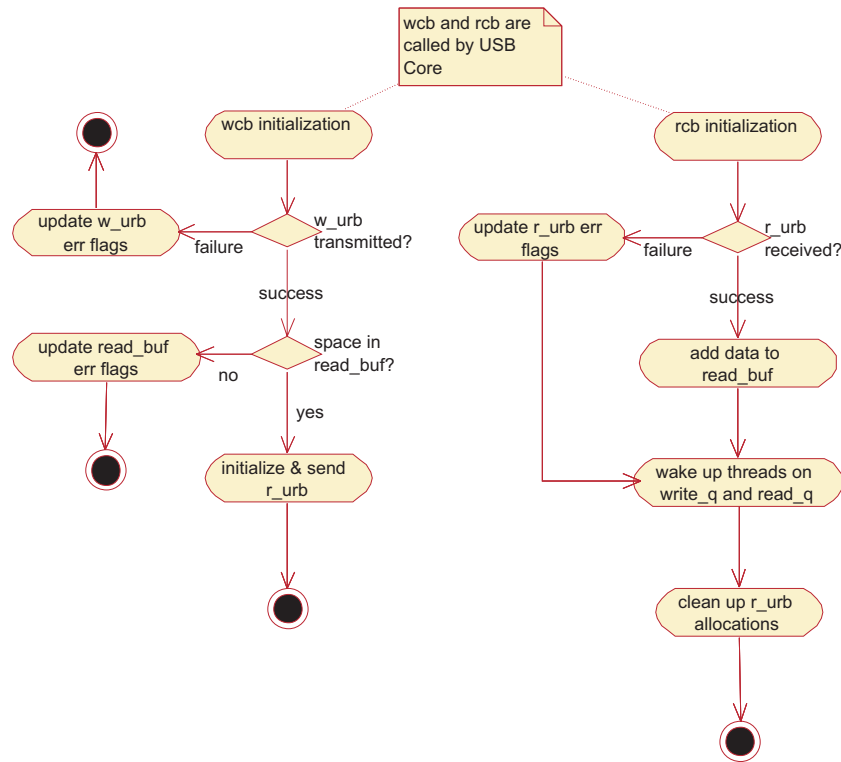


Figure 4.8: Activity diagram showing the flow of events involved in callback functions of the write thread.

states. In simple terms, packets with `CMD_SEND_REQ` headers are buffered, and the packet with the `CMD_FLUSH` header causes the execution enter into flushing state. Once the driver is in this state, it firstly checks if there is enough space in the `read_buf` and the thread is suspended in case of insufficient space. Since it is not possible to foresee how many bytes will be read, a heuristic value is empirically determined as a threshold, and any buffer having more free space than this threshold is considered as *available*. If the thread is able to continue executing, a urb to transmit the data to the device is initialized, which can be referred to as a *write urb* (`w_urb`), and submitted to the USB core, suspending the write thread until being signaled by one of the completion handlers. Upon wakeup, the function returns to the caller with the number of bytes successfully transmitted to device.

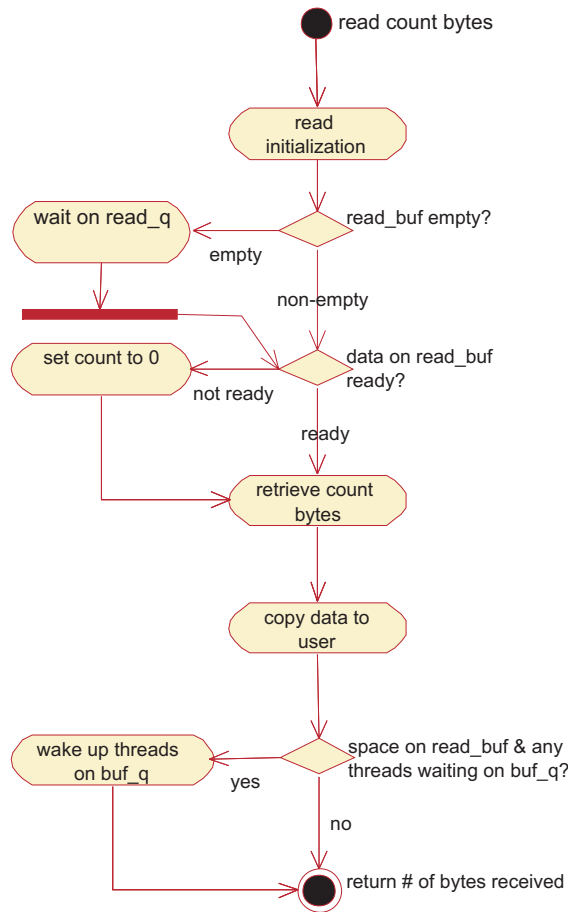


Figure 4.9: Activity diagram showing the flow of events involved in the read function.

4.4 URB Control Software

The URB Control Software is the central authority that is responsible for controlling the distributed system. It is an executable that runs on the host and consists of the user application, USB libraries and other system specific libraries.

4.4.1 USB Library

The USB Library is the USB instantiation of the URB CPU API. It consists of the `USBManager` implementing `BusManager` functionality for USB uplink implementations. The library relies on the presence of a kernel driver to handle

low level USB communications, implementing a specific communication protocol through a character device, depicted in Sections 4.1 and 4.3.4.

The **USBManager** is a multi-threaded **BusManager** instance whose operation involves two threads: **sendThread** and **readThread**. The **sendThread** is responsible from monitoring request lists and sending new requests to the bridge as they arrive. Furthermore, it also checks for the timer signal and periodically wakes up to send a flush command to the bridge, triggering transmission of buffered data within the driver, as well as a read of responses from the bridge. In contrast, the **readThread** continuously tries to read from the character device and processes incoming responses from the bridge. Based on various flags in the response, it matches them to previously sent requests and calls their callback functions if necessary.

The **USBManager** internally maintains four lists of requests: new requests, normal requests, urgent requests and auto-fetch requests. The first one keeps track of requests newly submitted through the **sendRequest()** method. In order to process a new request, **sendThread** transfers it to one of the remaining three lists based on its type. The requests remain in the lists until all responses associated with those requests are successfully received by the **readThread**.

It is also important to note that the **sendThread** must simultaneously wait for two different sources of events: the presence of a new incoming request and a timer signal. Both of these events use the request semaphore to inform the **sendThread**. If the **sendThread** detects that the new request list is empty after being able to lock the semaphore, it will conclude that the event source was the timer and issue a flush request.

4.4.2 User Application

The user application is a user level process where high level control algorithms are executed. It is linked to various libraries so as to fulfill the requirements of the system. For instance, the user application of the hexapod called **RHex** [24],

needs to link to the **RHexLib** library for behavioral implementations, i.e. walking, running with alternating tripod gait etc, and to the **URB** for the internal communication between itself and the distributed components. After linking to the appropriate libraries, the application uses facilities provided by these libraries to control the entire system.

In this section, we are only interested in how user application uses services of the URB uplink in order to communicate with distributed components of the system via a USB connection. Details of available services are explained in Sections 3.4.1, where the CPU API is described, and 4.4.1 where the USB instantiation of the API is summarized.

The user application first creates an instance of **USBManager**, which is a derived instance of the base class **BusManager** for USB connectivity. It then registers the **USBManager** object to the singleton instance of the **URBInterface**, by using its **addBusManager()** method. Afterwards, nodes within the system are queried by calling the **findNode()** method of **URBInterface**, based on specific class and index values supplied within the application. The methods of the **NodeAccessor** object that is associated with the found node are used for creation of **NodeRequest** object. The **NodeRequest** object has methods that allow allocation, initialization, and submission of the requests, as well as setting a callback function for the request, that is invoked upon the retrieval of the response from the node. The callback function can retrieve and process response data according to the particular needs of the control software. It is also possible to submit repeated node request objects, which will cause the associated callback to be invoked periodically as new responses are received. As a result, there are three possibilities for a request with this implementation, which are *write requests* to send data to the node inboxes, *read requests* to receive data from the node outboxes, and *repeated read requests* to receive data from the node outboxes periodically. The simple code snippet in Appendix B.3 illustrates how a user application can be implemented, which might clarify the concepts explained in this section.

Chapter 5

Performance Analysis of the USB Uplink

This chapter first presents a model that is suitable for the analysis of USB throughput characteristics, and then explains details of experiments conducted on the USB uplink for the URB, and presents their results. In general terms, our aim is to discover the USB characteristics of our hardware, estimate the limits on USB communication in terms of available bandwidth, round-trip latency, and scalability that can be achieved with our experimental settings, and finally test whether we can really achieve these communication estimates in practice.

5.1 Experimental Background

5.1.1 Analysis Model

In [14], a useful example on the analysis of USB throughput characteristics is presented, based on [7]. This analysis focuses on finding out how many bytes can be transmitted within the 1ms frame of the full speed mode. The analysis model is appropriate for our requirements, therefore, it will be shortly summarized in this section and adopted throughout this chapter.

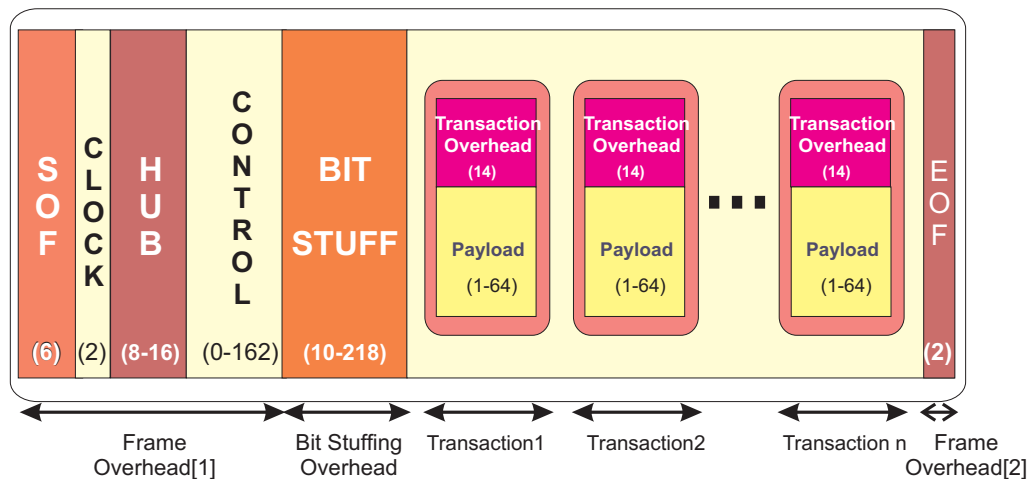


Figure 5.1: Structure of a frame with bulk transactions as a data block.

At full speed, a maximum of 1500 bytes can be transmitted within a USB frame. Figure 5.1 shows this arrangement for a frame of 1500 bytes, even though a frame is actually a time period. With this assumption, a frame can be considered as a stream of consecutive transactions, each transaction embodying a certain amount of payload (Refer to Section 2.4.2 for details). Moreover, since we are dealing with bulk transfers, we can make another assumption that a block of data to be transmitted between the host and a device is split into 64 byte chunks by the host controller driver and submitted to the target within consecutive transactions in the same frame as long as there is available bandwidth [14]. When we obtain experimental results, we can verify whether this assumption is appropriate or not by observing whether a data block larger than 64 bytes can be transmitted within a single frame duration. Note that it is normal for a certain part of this frame to be reserved for overhead, including bus (frame), bit stuffing and transaction overheads.

Bus overhead is the number of bytes required for frame synchronization and is composed of fields such as SOF (start of frame), EOF (end of frame), clock adjustment, hub polling, and bytes reserved for control transfer. The actual bus overhead might be much less than Figure 5.1, since a frame might not require any control transfers, or no hub change might have taken place in that frame.

Bit stuffing is used by USB to ensure the presence of sufficient signal transitions for clock recovery, and takes no more than 0.8% of total data on average. Although the worst case percentage is 16.67%, we do not consider this value in our calculations on the grounds that bulk transfer has a guaranteed delivery, and the transfer is retried if there is an error.

The last source of overhead is the transaction overhead, which is a result of token, handshake packets, inter-packet delays, and bus turnaround times. Figure 5.1 shows this overhead as 14 bytes per transaction, which is an estimate for bulk transfer at full speed.

Taking into account all these overheads, we can make an estimation on remaining bandwidth that can be used for bulk transfers. For the best case, we can assume that no control transfer or hub change takes place in that frame as well as no bytes are lost for bit stuffing. Since a bulk transaction of 64 byte payload requires 14 bytes of transaction overhead, we can calculate the number of transactions that fit into a frame as:

$$\begin{aligned}
 \text{Frame overhead} &= 6(\text{SOF}) + 2(\text{clock adjustment}) \\
 &+ 8(\text{hub query}) + 2(\text{EOF}) = 18 \\
 \text{Available size} &= 1500(\text{max size}) - 18(\text{frame overhead}) \\
 &= 1482 \\
 \text{Transaction size} &= 64(\text{payload}) + 14(\text{transaction overhead}) \\
 &= 78 \\
 \text{Num of transactions} &= \text{Available size} / \text{Transaction size} \\
 &= 1482 / 78 = 19
 \end{aligned}$$

In the worst case, we can assume that a control transfer and hub change takes place in that frame, as well as a bit stuffing of 0.8% (due to guaranteed bulk transfer) and calculate the number of transactions that fit into a frame as:

$$\begin{aligned}
\text{Frame overhead} &= 6(\text{SOF}) + 2(\text{clock adjustment}) + 16(\text{hub change}) \\
&+ 2(\text{EOF}) + 162(\text{control}) = 188 \\
\text{Bit stuffing size} &= (1500 - 188) * 0.008 \\
&= 10.5 \approx 11 \\
\text{Available size} &= 1500 - (188 + 11) \\
&= 1301 \\
\text{Num of transactions} &= \text{Available size} / \text{Transaction size} \\
&= 1301 / 78 = 16.7 \Rightarrow 16
\end{aligned}$$

Therefore, we may conclude that between 16 and 19 transactions can fit into a frame, which limits the actual bandwidth of USB bus to 8.2-9.7 Mbps for our case.

5.1.2 Analysis Goals

Our experiments are primarily based on measuring two important metrics: accumulation of unprocessed requests and round-trip latency.

Accumulation of unprocessed new request packets at the application is a hazardous situation since a continuous accumulation results in growth of the memory space reserved for the process. This situation might consume all free memory in a certain time period, whose duration is dependent on the available physical memory and accumulation rate, and the system which has run out of memory might hang entirely. Therefore, it is a fundamental requirement to determine the maximum load of the USB uplink such that the CPU successfully communicates with the bridge without any memory growth. As a result, we measure the number of unprocessed requests in order to determine the memory growth, and verify the growth by observing the increase in memory usage percentage of the process while it is in execution.

Round-trip latency is measured at the application level between the submission of a flush command to the driver by the sender thread of our USB library, which results in submission of buffered requests to the driver, and reception of the responses by the receiver thread of our USB library. It is important to remember that according to the USB uplink driver-to-application subprotocol, an invocation of a write system call for the flush command initiates an I/O operation that involves both submission of requests and receipt of responses. This I/O is a blocking operation, where the calling application thread is suspended until all responses are received at the kernel read buffer. Therefore a write call with flush command first sends data to the bridge in a USB frame, and then receives the response in the following frame, all of which is expected to last 2 ms for our test setting with USB in full speed mode. For our latency measurements, it is more important to obtain timing data with few outliers¹ than small average latency, since our challenge is to provide real-time guarantees for a 2ms uplink latency.

Another important issue that needs to be emphasized is that since timing measurements are done at application level, timing data includes the duration of context switching and other operating system overhead. Actually, the ideal method is to measure durations at kernel level. However, it is not feasible to obtain and interpret large amounts of data at kernel level. Therefore, we first obtained a small set of latency times both at kernel and application levels, and then concluded that we can normalize the application measurements to kernel level by subtracting 30us from raw application data. hence, all of the results in the following sections are normalized latency values and can be considered as kernel level measurements.

Although application level latency measurement with normalization is a rather reliable method of throughput analysis, we also verified the results of our test software with a Linux tool called `usbmon` [31], which runs at kernel level. Further details of the `usbmon` tool are available in Section 5.1.3.

Last goal of our experiments is to test whether the uplink is scalable or not, i.e. whether the performance of the system improves proportionally to the number

¹An outlier for this case is defined as timing data with latency higher than 2ms.

of connected bridges. This can be measured by gradually increasing the number of connected devices, and observing if the system still performs well in terms of latency and memory usage.

5.1.3 Test Software

We tested the USB uplink for the URB in two stages: testing with our own test software, and verifying the results with `usbmon`.

Our test software is composed of a shell script which gives us the opportunity to dynamically configure test parameters, together with the application software linked to our USB Library. The code snippet in Appendix B.3 shows the basic algorithm of our application, where it can be seen that a `USBManager` object is created and registered for each connected bridge. Virtual nodes connected to each bridge are then discovered based on their class and index values. Afterwards, there is a main loop which iterates as many times as requested, submitting multiple read and write requests within individual iterations, each of which is forced to take 2ms. More details on this application can be found in Section 4.4.2.

The test application is expected to determine the limits of communication. Therefore, data needs to be transmitted between the host and the bridge in a controlled manner. Since the maximum size of a URB packet is 33 bytes, including URB specific protocol wrappers, the main loop of our application iterates through virtual nodes on each bridge connected to the host, sending maximum sized read or write request packets within each iteration.

While the test application is in execution, it keeps a record of round-trip latencies for each flush operation, and outputs them into a text file. The application also displays various statistics about the communication, including the number of unprocessed requests and callbacks invocations as well as total transfer and execution times.

One final remark about our test application is that data read from the firmware includes a sequence number of the packet, which is then compared with a local

counter in the application to ensure no packet is lost. A similar application level transmission check could be implemented for the write requests as well; but we assumed that if there are no lost packets for the read case, there will also be no lost packets for the write case. Even then, we could detect that lost packet in our next step.

The verification step of the testing stage is realized with the Linux module `usbmon`, which operates as a kernel level USB sniffer, giving the opportunity to analyze data transmitted between the USB host controller and the device driver. The tool outputs transmitted USB traffic with a specific format, defined in [31]. We implemented a parser to process the output of `usbmon` into a format that satisfies our needs. The parser takes the output of `usbmon`, interprets it and outputs a text file which is used for plotting a graph illustrating size of transferred data versus time, as well as displaying information about total size of transferred data, total time, throughput and number of outliers in terms of latency. Results of the parser are then compared with our test application results in order to ensure that our test procedure was correct.

5.1.4 Mathematical Background

In this section, we define relations between the number of nodes, payload size, firmware constraints, throughput and number of transactions per frame. These abbreviations that will be used in the equations are as follows:

P_l : minimum payload size for a URB uplink packet

P_h : maximum payload size for a URB uplink packet

n : number of nodes

n_w : number of write requests submitted per 2 frame quanta

n_r : number of read requests submitted per 2 frame quanta

S_{rb} : size of receive buffer of firmware

S_{sb} : size of send buffer of firmware

Bw : throughput in Mbps

T : number of transactions within a frame

S_{bulk} : payload size of a USB bulk transaction

The firmware is implemented according to careful design decisions (Section 4.2), and our tests depend on certain features of the firmware, such as the transfer type, buffer sizes etc. The maximum bandwidth that can be obtained from the uplink directly depends on the size of firmware buffers that exchange data with the host.

Since requests and responses are exchanged between the host and the bridge in consecutive frames, we can assume that requests are sent to the device in odd numbered frames, and responses are received in even numbered frames. Write requests contain all data to be sent to the bridge, so they can be considered as having maximum size in odd frames. Similarly, responses to these write requests are just acknowledgment packets, so they have minimum size in even frames. The exact opposite is true for read requests, having minimum size in odd and maximum size in even frames. With this assumption we can express the relationship between the number of nodes and firmware buffer sizes as follows:

$$3 + P_h \times n_w + P_l \times n_r \leq S_{rb} \quad (5.1)$$

where frame number is 1,3,5 ... (odd)

$$P_l \times n_w + P_h \times n_r \leq S_{sb} \quad (5.2)$$

where frame number is 2,4,6 ... (even)

For our setting, values of fixed variables, which are determined in Chapter 4, are as follows:

$$P_l = 3, P_h = 33, S_{rb} = 512, S_{sb} = 1024$$

Substituting and simplifying these values into Inequalities 5.1 and 5.2 yields:

$$11 \times n_w + n_r \leq 169.7 \quad (5.3)$$

$$n_w + 11 \times n_r \leq 341.3 \quad (5.4)$$

So, values of w and r must satisfy Inequalities 5.3 and 5.4, as well as being natural numbers. If n_w equals n_r , then the inequalities are simplified to $n \leq 14$, which constraints the number of nodes to be between 1 and 14 inclusive in our tests. Any value that is outside this solution set will result in accumulation of requests at the host, and might cause unpredictable behavior. Therefore, we will not be considering outside of this range in our tests. Note that, this derivation, and all the following equations, assume that only one message box per node is used. We know that there are 6 more message boxes per node that are available for the use of a URB user.

Different values of n give the opportunity to test different bandwidths. The relation between the number of nodes and data throughput in Mbps can be found as follows:

$$Bw = \frac{8 \times (P_l + P_h) \times n}{1000} \quad (5.5)$$

which can be simplified to following equation after substituting $P_l = 3$ and $P_h = 33$ for our experimental setting:

$$Bw = \frac{36 \times n}{125} \quad (5.6)$$

where 3-bytes of flush packet is ignored for making the throughput follow a constant rate over time.

Finally, number of transactions that fit into a frame is also an important parameter, whose relation with the number of nodes can be calculated as follows:

$$T = \lceil \frac{(P_l + P_h) \times n}{S_{bulk}} \times k \rceil \quad (5.7)$$

where k is an empirical constant whose value is 1 for theoretical computation of transaction size, under which circumstances it is assumed that no extra host delay is existant along the physical USB bus.

For our experimental setting, the values of fixed variables are as follows:

$$P_l = 3, P_h = 33, S_{bulk} = 64, k = 1.017$$

Substituting these values into Equation 5.7 yields

$$T = \lceil 0.572 \times n \rceil \quad (5.8)$$

5.2 Experimental Setup

In this section, properties of hardware and software used in our test URB system are given. A summary of the test environment is as shown in Table 5.1.

	URB CPU	URB Bridge
Type	IBM compatible PC	Silabs F340 board
Processor	Intel Core2Duo 1.83GHz	8051-compatible CIP-51 Core
Memory	Kingston 1GB RAM	4352-byte data RAM and 64KB flash
USB Controller	UHCI	USB0 module
Operating System	Linux (Ubuntu Intrepid, 2.6.29 kernel, 1000Hz timer)	None
IDE	Emacs and gcc/g++ (Linux)	Keil MicroVision3 (cross-compilation from WindowsXP)

Table 5.1: Test Environment Settings

5.2.1 Configuration of the URB CPU

In all of our experiments, the host operating system kernel is configured to operate at a higher frequency by modifying the default system timer to run at 1000Hz. After successful compilation and installation of a kernel with the new timer frequency, we were able to generate interrupts at a period equal to the USB Host Controller frame period of 1ms.

Since we are interested in precise measurement of uplink characteristics in terms of latency and memory usage, it is very important to eliminate noise which interfere with the results in our test environment. Therefore, XWindows and any

other processes which are not expected to run in the target platform, i.e. the robot, are terminated on the host before the experiments are started.

5.2.2 Configuration of the URB Bridge

The bridge is configured to operate at 48MHz clock by the firmware, which is a requirement for the full speed mode of F340 board. There are two versions of bridge firmware, which are firmware with loopback downlink interface, and firmware with I^2C downlink interface. The former is a trivial firmware to be used in uplink tests, and behaves as if virtual nodes are connected to it on the downlink, responding to any CPU node requests with dummy data, encoded with only a sequence number, and the node address. The latter is the real firmware that is to be run in a full URB system, and it forwards any node requests to appropriate real node, receives the response from downlink and forwards to uplink channel. The firmware used in the tests of this section is the loopback downlink, since we are only interested in the USB uplink characteristics of URB in this study.

5.3 Estimation of Host USB Characteristics

Hardware design of the test platform for USB support is an important issue that needs to be carefully investigated in order to carry out experiments reliably. We used the tool called USBView and Linux commands `lsusb` and `lspci` for an estimation on USB hardware design on our host PC. As a result, we concluded that there are five HCs on our host, four of which are UHCI for USB1.1 support and the last one EHCI for supporting USB2.0 devices. The PC has 4 USB ports, and when USB1.1 devices are considered, these ports are controlled by two of the UHCI HCs, 2 ports per each UHCI. The third UHCI HC is used for integrated Bluetooth support and the last UHCI is not observed to control any devices with current PC settings. Lastly, the integrated webcam is controlled by EHCI.

`lsusb` output informs us that there are 5 USB buses on the PC, one for each HC, and we used 2 buses connected to UHCI HCs for our experiments, and the additional high speed bus for the experiments with the external USB hub. We used `usbmon` to sniff other buses while carrying out experiments and found out that devices on other buses i.e. the webcam, bluetooth, have no interference to our results since they are controlled by different HCs.

5.4 Experiments and Results

In this section, our experimental methods are explained, their results are presented and discussed. Our experiments proceed in two steps, single bridge tests and multiple bridge tests.

5.4.1 Single Bridge Tests

In this experiment, a single bridge is connected to the host, and the test application is executed to iterate 1000 times, sending out maximum sized write and read requests to connected virtual nodes. The number of connected virtual nodes is increased from 1 to 14 one by one, and the application is executed once for each value of virtual nodes. Consequently, since size of transferred data increases proportional to the number of nodes, we are able to search for the maximum possible data rate with this approach.

During the execution of the test application, the appropriate USB bus is sniffed with `usbmon` for obtaining more precise results, as mentioned in Sections 5.1.2 and 5.1.3. The output of `usbmon` is then given to a parser to produce similar statistics to the those of the application. Table 5.2 shows a comparison between the results of the application and `usbmon`, where it can be seen that the two are consistent. On average, total execution times and transfer sizes obtained from `usbmon` are 0.3% and 0.01%-1.00% more than the application measurements respectively. Recall that results related to timing at the test application

n	Total Transfer Size(bytes)		Total Execution Time(ms)	
	application	usbmon	application	usbmon
1	75288	75297	2195	2198
2	147318	147333	2214	2226
3	219321	219327	2207	2214
4	291333	291342	2215	2220
5	363309	363321	2202	2208
6	435288	435306	2199	2202
7	491292	492981	2214	2221
8	451053	454017	2215	2224
9	497052	499584	2226	2238
10	524085	526536	2233	2242
11	477432	481485	2235	2249
12	528624	532668	2240	2254
13	543042	548076	2246	2264
14	499641	504744	2254	2274

Table 5.2: Comparison of data from usbmon and application.

are normalized to kernel level as mentioned in Section 5.1.2. The difference in time measurement is possibly a result of this normalization, which reveals that we should have subtracted less than 30 microseconds from original results. The difference in size measurement is probably due to the fact that we do not have a method to measure exact number of bytes at the application. Instead, we estimated the size according to the number of callbacks received by the application. Since direct kernel measurement is more reliable than statistical normalization or mathematical estimations, we will continue our analysis with only results from *usbmon*, which are expected to be more reliable.

Figure 5.2 illustrates the change in the number of unprocessed requests as a function of the number of nodes according to the output of *usbmon*. The number of unprocessed requests is zero for values of n from 1 to 6. It then starts increasing rapidly. The rapid increase is an indication of the accumulation of unprocessed requests, resulting in memory growth. When the value of n equals 6, number of transactions per frame is calculated as 4 by (5.8).

The meaning of this accumulation is that the USB device with the current firmware is not fast enough to sustain more than 4 repeated full size bulk transactions within a frame. If we could measure the delay between the transactions

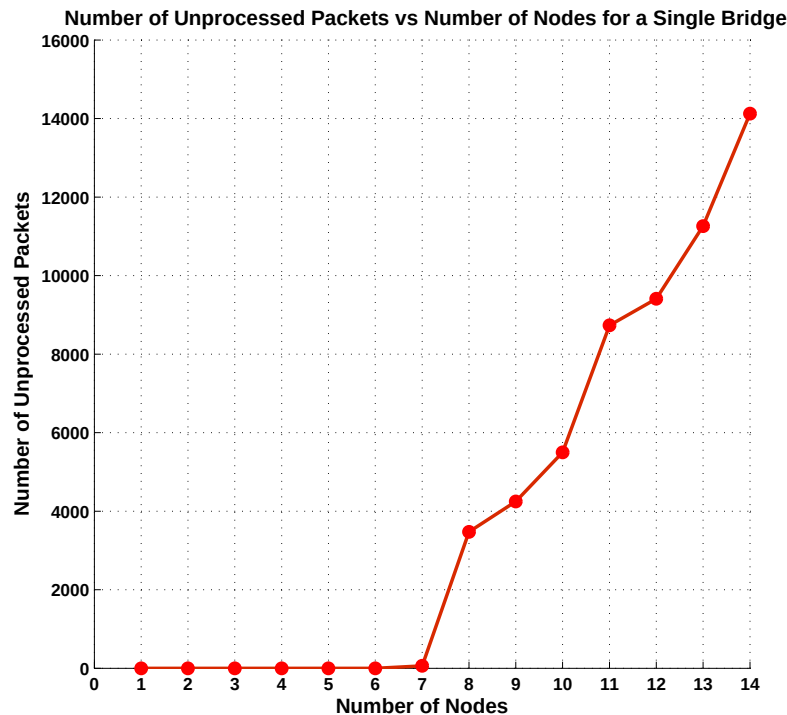


Figure 5.2: Change of number of unprocessed requests over number of nodes based on usbmon output for 1000 request submissions. Number of unprocessed requests is 0 for $n=1:6$.

with a bus analyzer, we would obtain an estimate of the data rate the system could have achieved if the device was able to keep up.

Although communicating with 6 nodes theoretically yields a throughput of 1.73Mbps, from (5.6), the parser reported a *net throughput* of 1.57Mbps. The value obtained from the parser is more reliable, since theoretical estimation is making the assumption that there is a constant rate of transfer, without any fluctuations. However, the real situation is different, as change of transfer size for 3 transfers with $n:1,6,14$ over time are shown in Figure 5.3. The transfer rate shows fluctuations, most of which are down to 3 bytes. Such a sudden decrease is an indication of an empty flush submission, which occurs when timer invokes a flush command although there is nothing in the kernel write buffer. The reason behind this event is that each submission of requests is followed by a blocking time equal to the flushing period, 2ms, at the application, in order to

prevent flooding of requests. Since submission of requests also take some time in the order of microseconds, accumulation of these microsecond periods result in such empty flush requests when their total value exceeds 2ms. Figure 5.3 is also useful for comparison of communication at different bandwidths. We see that the pattern of transfer where n equals 14 is more irregular than the others, which is an expected situation since our settings do not reliably support communication at such high bandwidths.

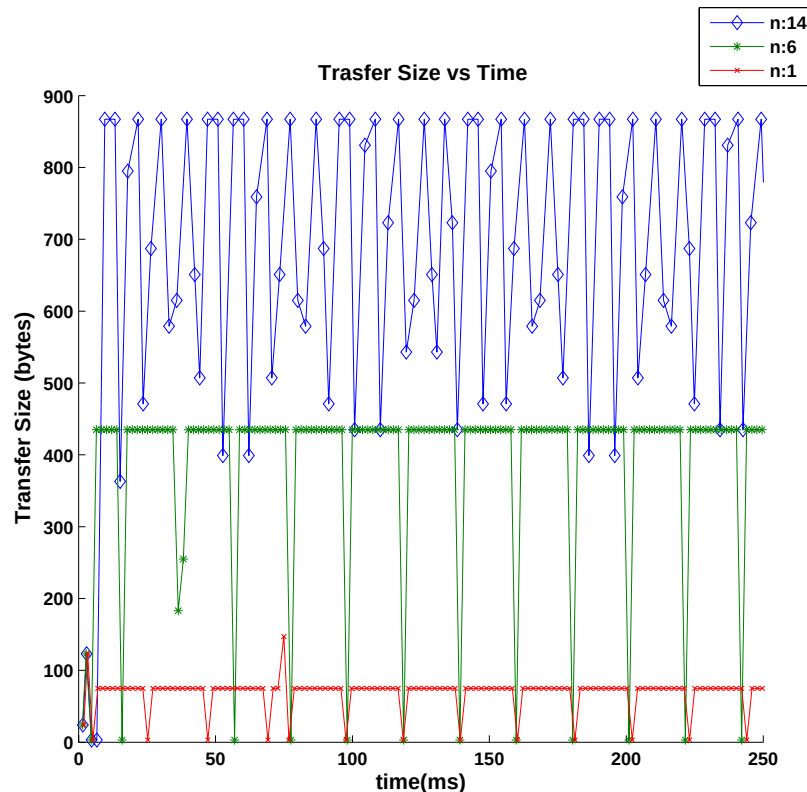


Figure 5.3: Comparison of time versus transfer size for 3 transfers with $n:1,6,14$, based on usbmon output.

Another reason for the practical net throughput being less than the theoretical throughput is the effect of the time elapsed for the execution of context switches, overhead of other running processes and operating system, blocked state of 2ms and so on. The parser reported a *transfer throughput* of 1.70Mbps when only the time for transfers were considered, which is much closer to the theoretical value.

We can infer from these discussions that with our USB uplink protocol, a

single device can reliably support a maximum data rate of almost 1.60 Mbps, though it could sustain a bit more than that if we had increased the resolution by decreasing the value of maximum payload, which would increase n in turn. This throughput is achieved with 6 virtual nodes in our settings, and for any value of n above 6, the results are nondeterministic, showing either a fluctuation in terms of latency, number of outliers, transferred data size, or a nonlinear increase in terms of unprocessed requests.

Note that 6 is an upper bound for n only under current experimental settings where maximum sized URB bulk transactions are transmitted, and one message box per virtual node is used. The fundamental constraint is not the number of nodes, but the number of transactions, obtained from 5.7. One can change the number of nodes by modifying the value of P_h between 3 and 33 as long as T is constant at the maximum supported value. So for our case, we can change the values of P_h and n as long as 5.7 yields 4 transactions.

After pointing out this important note, we can continue the analysis for the interval where values of n between 1 and 6, and the remaining range of n is out of our scope from now on.

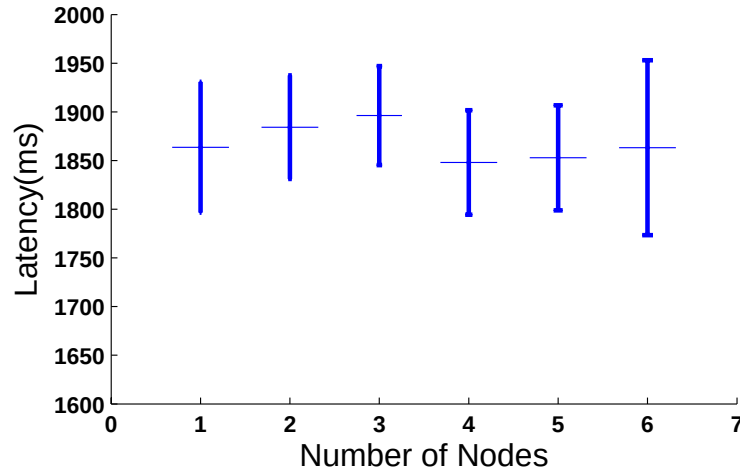


Figure 5.4: Mean and standard deviation values of round-trip latencies for 1000 request submissions to each node, obtained from `usbmon`. Note that y-axis of the plot is scaled between 1600-2000ms in order to display the standard deviations more clearly.

n	mean(ms)	std(ms)	min(ms)	max(ms)	outliers
1	1.864	0.067	1.113	2.098	6
2	1.884	0.053	1.079	2.968	2
3	1.896	0.051	0.908	2.030	1
4	1.848	0.054	0.874	1.994	0
5	1.853	0.054	1.165	2.612	4
6	1.863	0.090	1.169	3.595	7

Table 5.3: Mean and standard deviation values of round-trip latencies for 1000 request submissions to each node, obtained from *usbmon*.

Figure 5.4 displays the mean and standard deviation values of round-trip latencies for each node. Each value is obtained by extracting round-trip latencies from the output of *usbmon* for the submission of 1000 requests. These results are also summarized in Table 5.3, as well as min-max values of latencies, and number of outliers.

We can infer from Figure 5.4 and Table 5.3 that since average latencies and number of outliers show a fluctuation, the value of latency is not directly related to the size of transfer as long as the number of transactions transmitted within a frame is sustained by the device. The value of latency is more dependent on the load of operating system, which determines the real-time response capability of it. Although the percentage of outliers is as low as 0.1-0.9%, we need to ensure that there are no outliers for the supported net throughput up to 1.60 Mbps in order to claim that we truly guarantee real-time operation. However, since Linux is not a RTOS, it is acceptable to have so low percentage of outliers, which can only be completely eliminated by porting our system to a RTOS.

Figure 5.5 shows round-trip latencies of 1000 requests for $n=6$. As you see, after the first few transmissions, the latency values are synchronized and propagate along a line at about 1.85ms, showing mostly small fluctuations, together with a few outliers. This figure lets us to see the outliers more clearly.

As a consequence, we analyzed the characteristics of the communication with a single bridge through the USB uplink for the URB, and found out that a maximum bandwidth of approximately 1.60 Mbps is supported by the protocol.

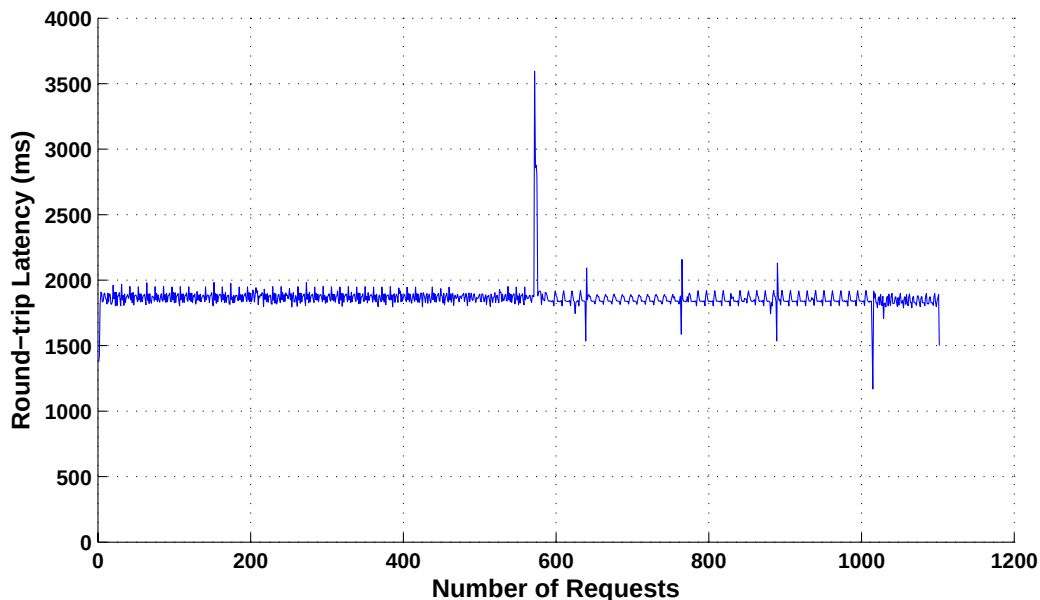
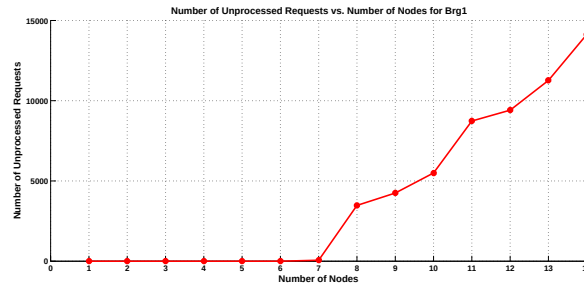


Figure 5.5: Round-trip latencies of 1000 requests for $n=6$, obtained from `usbmon`.

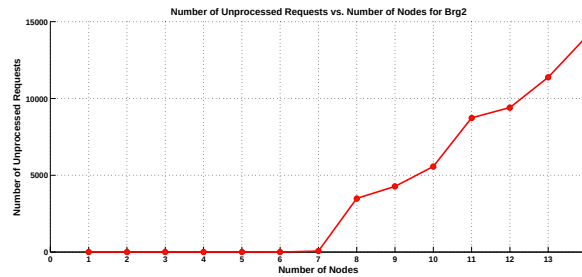
5.4.2 Multiple Bridge Tests

In this set of experiments, we connected multiple bridges to the host PC and repeated the experiments above for multiple devices. The first test case is done with two bridges connected to the same root hub. Since the two devices share the same root hub, the maximum bandwidth available, 9.7Mbps according to Section 5.1.1, is expected to be divided by two. However, since the maximum net throughput a device can sustain under current settings is obtained to be 1.60 Mbps in single device tests, it is not possible to observe if the bandwidth is really divided by two. Our aim for multiple bridge test is to see how the USB HC handles two devices concurrently, especially in terms of synchronization and timing between two bridges.

While Figure 5.6 shows change of number of unprocessed requests over number of nodes, Figure 5.7 displays mean and standard deviation values of round-trip latencies for 1000 request submissions to each node of `bridge1` and `bridge2`. As can be seen, results are almost the same as the single bridge test. Moreover, Table 5.4 shows a comparison between various transfer statistics of two bridges



(a)



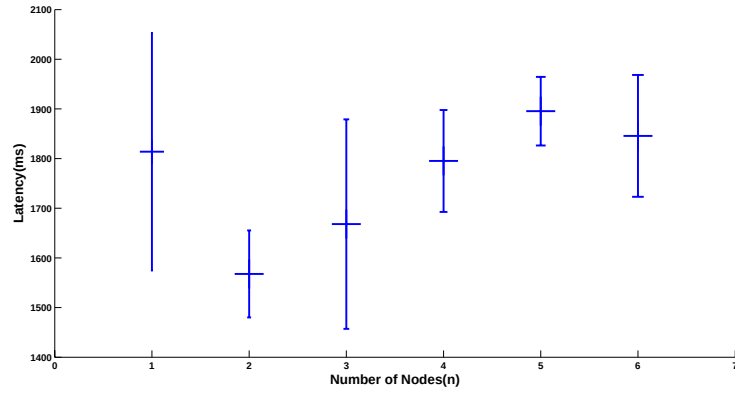
(b)

Figure 5.6: Number of unprocessed requests over number of nodes for (a) bridge 1. (b) bridge 2. Number of unprocessed requests is 0 for $n=1:6$ for both bridges.

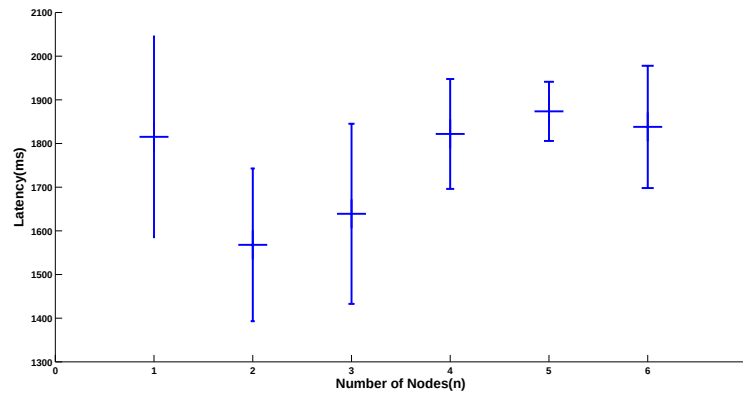
for n equals 6, where we see that results from both bridges are also very close to each other. From all these information, we may conclude that USB UHCI host controller successfully handles multiple devices connected to the same root hub, and supports up to 1.54 Mbps of data rate per bridge.

Next test case is done with three bridges, two connected to one root hub, and the remaining one to the other root hub. Table 5.5 shows a comparison between the results from three bridges for n equals 6. Once more, results are not different from previous tests. Therefore, we may conclude that USB UHCI HC successfully handles multiple devices connected to the different root hubs as well. Also, from the results of tests with 2 and 3 bridges, we observed that the bandwidth per bridge does not decrease as the number of bridges increases. Therefore, we can conclude that our system is scalable, supporting a total throughput of 5.114 Mbps with three bridges reliably, without any missed packets.

The main goal for these tests was to estimate the maximum bandwidth of a reliable USB uplink communication with our protocol. Therefore, all of these tests



(a)



(b)

Figure 5.7: Mean and standard deviation values of round-trip latencies for 1000 request submissions to each node of (a) bridge 1 (b) bridge 2, obtained from usbmon. Note that y-xis of the plot is scaled between 1300-2100ms in order to display the standard deviations more clearly.

were done considering worst case scenarios, such as maximum sized data requests and responses. Under normal circumstances, a small-scale rapid autonomous robot does not need to communicate at such high bandwidths with so extensive traffic. Also, since our I^2C -based downlink protocol operates at a maximum data rate of 1 Mbps, obtaining a USB uplink throughput of 1.60 Mbps per bridge is far enough for our initial goals. Also, since our uplink is found to be scalable, the number of nodes can be increased upon requirement by increasing the number of connected bridges.

	Bridge1	Bridge2
Total transfer size (bytes)	435345	435495
Avr of Latency (ms)	1.846	1.838
Std of Latency (ms)	0.123	0.140
Number of outliers	8	6
Net Throughput (Mbps)	1.554	1.489

Table 5.4: Various transfer statistics of two bridges for n=6 and 1000 request submissions, obtained from usbmon.

	Bridge1	Bridge2	Bridge3
Total transfer size (bytes)	435378	435378	435528
Avr of Latency (ms)	1.809	1.687	1.881
Std of Latency (ms)	0.162	0.157	0.048
Number of outliers	6	5	0
Net Throughput (Mbps)	1.545	1.543	1.480

Table 5.5: Various transfer statistics of three bridges for n=6 and 1000 request submissions, obtained from usbmon. Bridges 1 and 2 are connected to one root hub, and Bridge 3 is connected to another root hub.

Chapter 6

Conclusion

In this thesis, we described a USB based real-time uplink connectivity for the URB [5], which is a distributed control architecture for interfacing local nodes of a mobile robot platform [4]. We were motivated by the problem that the RS232 uplink performance of the URB was poor due to low bandwidth, lowering the communication performance of overall system. We evaluated alternative communication standards that could be incorporated to URB as an uplink channel so that communication between the URB CPU and the URB bridge could be established at a high bandwidth. The best candidate was Universal Serial Bus, which provided many advantages such as high and guaranteed bandwidth, hot-plugging, easy and low-cost physical connectors.

Although there are a few problems of USB which obstructs its wide usage in real-time applications, the advantages of its high bandwidth traded off its drawbacks in our work. The first issue was its lack of interrupt support, which did not cause problems to us in terms of real-time communication. We were able to define an upper bound of 2ms on uplink round-trip latency at a bandwidth of almost 1.6 Mbps per bridge, much higher than the downlink I^2C bus could handle which was 1Mbps. Moreover, USB had a bad reputation in synchronization and timing of multiple devices, however, we observed in our experiments that there was no problem in communication with multiple bridges at 1.6 Mbps.

As a consequence, we implemented a USB uplink for the URB, and observed that it successfully handles uplink communication at a high bandwidth with an acceptable latency. A very low percentage of outliers was observed in our experiments, which revealed that our current uplink implementation does not truly provide a guarantee for a 2ms latency. We need to use a RTOS so as to totally eliminate these outliers, and provide a true real-time uplink to the users. We are working on porting USB drivers to QNX operating system, which will ensure real-time connectivity.

Bibliography

- [1] TCNET. <http://www3.toshiba.co.jp/sic/english/seigyotcnet/index.htm>.
- [2] USB Host Stack white paper. <http://www.jungo.com>.
- [3] K. Akachi, K. Kaneko, N. Kanehira, S. Ota, G. Miyamori, M. Hirata, S. Kajita, and F. Kanehiro. Development of humanoid robot hrp-3p. In *Proceedings of the 5th IEEE-RAS International Conference on Humanoid Robots, 2005*, pages 50–55, Dec. 2005.
- [4] A. Avci. The Universal Robot Bus: A local communication infrastructure for small robots. Master’s thesis, Bilkent University, Computer Engineering Department, 2008.
- [5] A. Avci and U. Saranli. URB detailed design document. Technical report, Bilkent University, Computer Engineering Department, 2007.
- [6] R. Cleaveland and S. A. Smolka. Strategic directions in concurrency research. *ACM Computing Surveys*, 28:607–625, 1996.
- [7] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips. *USB1.1 Specification*.
- [8] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips. *USB 2.0 Specification*, 2000.
- [9] Electronics Industries Association. *EIA Standard RS-232-C Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Data Interchange*, August 1969.

- [10] D. R. Evoy, L. Goff, P. Chambers, and M. Eidson. Real time event determination in a USB system. *US Patents*, (5958020), 1999.
- [11] D. Fliegl. Programming guide for Linux USB device drivers. <http://usb.cs.tum.edu>, 2000.
- [12] P. Foster and C. Svensrud. Extending USB to time-critical applications. <http://www.sensorsmag.com/>, 2006.
- [13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [14] J. Garney. *An Analysis of Throughput Characteristics of Universal Serial Bus*. Intel Architecture Labs, 1996.
- [15] Y. Ishiwata and T. Matsui. Development of linux which has advanced real-time processing function. In *16th Annual Conference of Robotics Society of Japan*, 1998.
- [16] J. Jasperneite and J. Feld. PROFINET: an integration platform for heterogeneous industrial communication systems. In *Proceedings of the 10th IEEE Conference on Emerging Technologies and Factory Automation, 2005.*, volume 1, pages 8 pp.–822, Sept. 2005.
- [17] F. Kanehiro, Y. Ishiwata, H. Saito, K. Akachi, G. Miyamori, T. Isozumi, K. Kaneko, and H. Hirukawa. Distributed control system of humanoid robots based on real-time ethernet. In *Proceedings of the International Conference on Intelligent Robots and Systems, 2006 IEEE/RSJ*, pages 2471–2477, Oct. 2006.
- [18] H. Komsuoglu and A. A. Rizzi. *RISEBus Specification*. UMICH, CMU, 2004.
- [19] I.-W. Park, J.-Y. Kim, S.-W. Park, and J.-H. Oh. Development of humanoid robot platform khr-2 (kaist humanoid robot-2). In *Proceedings of the 4th IEEE/RAS International Conference on Humanoid Robots, 2004*, volume 1, pages 292–310 Vol. 1, Nov. 2004.
- [20] Philips Semiconductors. *The I2C-BUS specification*, 2.1 edition, January 2000.

- [21] Robert Bosch GmbH. *CAN Specification*, 2.0 edition, 1991.
- [22] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
- [23] A. Rubini, J. Corbet, and G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3rd edition, 2005.
- [24] U. Saranli, M. Buehler, and D. E. Koditschek. RHex: A simple and highly mobile hexapod robot. In *The International Journal of Robotics Research*, volume 20, pages 616–631, July 2001.
- [25] A. Saunders, D. I. Goldman, R. J. Full, and M. Buehler. The RiSE climbing robot: Body and leg design. In G. R. Gerhart, C. M. Shoemaker, and D. W. Gage, editors, *Unmanned Systems Technology VII*, volume 6230, page 623017. SPIE, 2006.
- [26] N. Sclater and N. Chironis. *Mechanisms and Mechanical Devices Sourcebook*. McGraw-Hill, 2001.
- [27] Silicon Laboratories Inc. *C8051F340/1/2/3/4/5/6/7 datasheet*, 2006.
- [28] K. A. Tahboub. A semi-autonomous reactive control architecture. *Journal of Intelligent and Robotic Systems*, Volume 32:445–459, 2001.12.01.
- [29] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
- [30] C.-P. Young, M. Devaney, and S.-C. Wang. Universal serial bus enhances virtual instrument-based distributed power monitoring. *IEEE Transactions on Instrumentation and Measurement*, 50(6):1692–1697, Dec 2001.
- [31] P. Zaitcev. The USBMON: USB Monitoring Framework. In *Linux Symposium 2005*, 2005.
- [32] C. Zhang, H. Wang, and J. Wang. An USB-based software CNC system. *Journal of Materials Processing Technology*, 139(1-3):286 – 290, 2003.

Appendix A

URB Downlink Details

A.1 Downlink Control with Inbox/Outbox 0

Downlink control is established by using Outbox 0 and Inbox 0 of the node, which are responsible for providing node configuration information and reception of URB related commands, respectively. URB protocol uses OutBox 0 of each node to provide information necessary to identify the functionality of the node. This outbox is read in the discovery phase of the initialization to build a catalog of which URB node addresses correspond to what kinds of functionality.

The type of functionality provided by each node is specified by its **NodeClass**. Different instances of nodes implementing the same functionality (such as multiple motor drivers connected to different motors in a robot) are distinguished by their **NodeIndex** values. These configuration parameters are selected by the node application during its initialization using specific methods provided by the Node API. Designers must ensure that no two nodes on the same downlink bus have the same **class/index** pair.

The **NodeVer** and **NodeRev** fields encode the version and revision numbers for the local firmware on the node. These fields can be retrieved by the control software on the CPU to ensure proper versioning of the functionality and message

box formats implemented by the node. The `NodeState` field encodes the current state of the URB subsystem on the node. The `C` bit specifies type of downlink transaction, and further details can be found in [5].

The URB protocol uses Inbox 0 of each node to send protocol commands to URB nodes. The opcode field encodes the command, among the alternatives of sending an asynchronous signal to the node, synchronization signal, enabling/disabling phase locking to synchronization messages etc. When data is written into Inbox0, the URB Node library decodes the command and performs the associated action. There is usually no need to directly access Inbox 0 from within user application code on the CPU [5].

Appendix B

URB Uplink Details

B.1 URB Uplink Bridge Commands

Table B.1: Bridge command types with description

Bridge Command Type	#args	#resp
BRG_RESET_CMD	0	N/A
Reset the bridge and all nodes. <i>Arguments:</i> NONE <i>Response:</i> NONE even if RR=1		
BRG_GETVER_CMD	0	2
Request bridge firmware version and revision. <i>Arguments:</i> NONE <i>Response:</i> Version followed by revision (each in one byte).		
BRG_CLOCK_CMD	0	4
Request current clock value of bridge.		
Continued on next page.		

Bridge Command Type	#args	#resp
<i>Arguments:</i> NONE <i>Response:</i> 4 byte integer with most significant byte first.		
BRG_LED_CMD	1	1
Sets the states of a specified LED located on each bridge. <i>Arguments:</i> 1 byte to define the LED number (MS 7 bits) and next state of the specified LED (LS 1 bit). <i>Response:</i> 1 byte to define the previous state of the LED.		
BRG_PKTCount_CMD	0	8
Request uplink and downlink packet counts. <i>Arguments:</i> NONE. <i>Response:</i> 4 byte integer with uplink packet count (most significant byte first), followed by a 4 byte integer with the downlink packet count (most significant byte first).		
BRG_DISCOVER_CMD	0	2
Request for discovery of nodes and respond with a report of active nodes. <i>Arguments:</i> NONE <i>Response:</i> 1 _{st} byte encodes availability of nodes 15-8, 2 _{nd} byte encodes availability of nodes 7-0. Active nodes are indicated with binary 1.		
BRG_AUTOMATE_CMD	1	1
Enables or disables autonomous synchronization messages and autonomous data fetch operations coordinated by the bridge and defines the frequency of the synchronization messages. <i>Arguments:</i> 1 bit for enable autonomous synchronization (MSB 1 for enable, 0 for disable), 1 bit to enable autonomous fetch operations (bit #6) and (6 bits) frequency of the heartbeat messages.		

Continued on next page.

Bridge Command Type	#args	#resp
<i>Response:</i> 1 byte of ACK is sent if RR is enabled(1 for success, 0 for failure).		
BRG_AUTOFETCH_CMD	2	1
Defines the nodes and message boxes to be read periodically after enabling autonomous fetch operation. <i>Arguments:</i> 1 byte to define node address (MS 4 bits) and message box ID (3 bits). 1 byte for outbox size that is to be fetched. <i>Response:</i> If RR=1 then 1 byte of ID is sent. This ID is used to identify the queued autonomous fetch operations.		
BRG_CANCEL_AUTOFETCH_CMD	1	1
This command is used to remove any previous autonomous fetch operations stored in the queue. <i>Arguments:</i> 1 byte with the ID of the fetch operation that is going to be deleted from queue or 0xFF to remove all the previously defined autonomous fetch operations. <i>Response:</i> If RR=1 then 1 byte of ACK is sent (1 for success, 0 for failure).		
BRG_NODEINFO_CMD	1	4
This command is used to retrieve node information of a particular node. Bridge responds to this command with the information of nodes after the last discovery. <i>Arguments:</i> 1 byte with node address. <i>Response:</i> 1 byte with the node address, followed by 3 bytes with the same format as outbox[0] encoding node information.		
BRG_NOP_CMD	0	1
Dummy command with no effect. Ignored by bridges. <i>Arguments:</i> NONE <i>Response:</i> If RR=1 a dummy response of 1 byte with value 0 is generated by bridges.		
BRG_TOKEN_CMD	0	4
In response to this command, bridges send a special 4 bytes token. Continued on next page.		

Bridge Command Type	#args	#resp
<i>Arguments:</i> NONE <i>Response:</i> 0xFFA5B99F (MSB first).		
BRG_ECHO_CMD	1	-
Directly echoes the argument through uplink. <i>Arguments:</i> 1 byte to be echoed. <i>Response:</i> Response to this command is not in standard packet format. Given argument directly echoes through uplink.		
BRG_DL_GETVER_CMD	0	10
In response to this command, downlink version, revision and type string is sent. <i>Arguments:</i> NONE <i>Response:</i> Downlink version (1 byte), revision (1 byte) and string encoding the type of the downlink (8 bytes).		
BRG_DL_CUSTOM_CMD	VAR	VAR
Send a custom, user defined command to downlink system. <i>Arguments:</i> Variable <i>Response:</i> Variable		
BRG_UL_GETVER_CMD	0	10
In response to this command, uplink version, revision and type string is sent. <i>Arguments:</i> NONE <i>Response:</i> Uplink version (1 byte), revision (1 byte) and string encoding the type of the uplink (8 bytes).		
BRG_UL_CUSTOM_CMD	VAR	VAR
Send a custom, user defined command to uplink system. <i>Arguments:</i> Variable <i>Response:</i> Variable		

B.2 URB Uplink Packet Formats

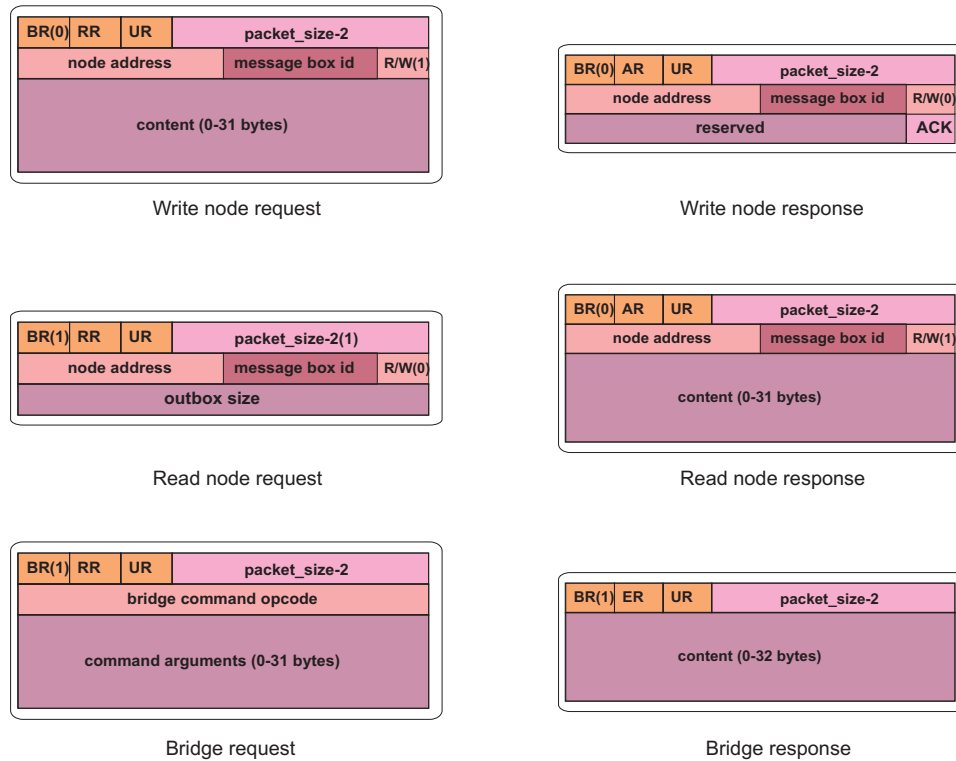


Figure B.1: URB Uplink Request and Response Packet Formats.

B.3 An Example User Application (CPU Side)

```
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include "urb/URBInterface.hh"
#include "urb/Request.hh"
#include "urb/USBManager.hh"
#include "urb/RS232Manager.hh"

#define REPEAT 1000
#define BUS_AUTOSYNC_CMD 0x07
#define MAX_PAYLOAD 31
```

```

/*time duration for communication*/
#define WAIT_MS 2

#define NUM_NODES 12
#define NUM_READS 6
#define NUM_WRITES 6

#define URB_REQ_RESPONSE 0xC0

static bool control_c_invoked = false;

void exit_on_ctrl_c(int)
{
    printf( "Caught Ctrl-C signal. Exiting main loop..!\n");

    if (control_c_invoked)
        return;

    control_c_invoked = true;
}

static bool read_callback( urb::Request *request ) {

    static int count = REPEAT;

    urb::NodeRequest *req = (urb::NodeRequest *) request;
    const urb::node_info_t *info = req->getNode()->getNodeInfo();
    int dataSize = req->getPacketSize() - 2;
    urb::byte * data = req->getPacket();

    if ( --count == 0 ) {
        printf( "[@read_cb %p] (bus:%d,n_adr:%d,mbox:%d): ",
                req, info->bus_id, info->address,
                URB_NODE_MBOX( data[1] ) );
    }
}

```

```

    for (int i = 0; i < dataSize+2; i++ ) {
        printf( "0x%02x ", data[i] );
    }

    printf("\n");
    count = REPEAT;
}

return true;
}

static bool write_callback( urb::Request *request ) {
    static int count = REPEAT;

    urb::NodeRequest *req = (urb::NodeRequest *) request;
    const urb::node_info_t *info = req->getNode()->getNodeInfo();
    int dataSize = req->getPacketSize() - 2;
    urb::byte * data = req->getPacket();

    if ( --count == 0 ) {
        printf( "[@write_cb]: Wrote %p: (%d,%d,%d): ", req, info->bus_id,
            info->address, URB_NODE_MBOX(data[1]) );

        for (int i = 0; i < dataSize+2; i++ ) {
            printf( "0x%02x ", data[i] );
        }

        printf("\n");
        count = REPEAT;
    }

    return true;
}

```

```

int main( int argc, char **argv ) {

    signal( SIGINT, exit_on_ctrl_c );

    urb::NodeAccessor *acc[NUM_NODES];
    urb::node_info_t node;

    urb::NodeRequest *req;

    urb::URBInterface *urbintf = urb::URBInterface::instance();
    urb::USBManager *usbm0 = new urb::USBManager(
        (char *) "/dev/urb_usb0" );

    int i;

    urbintf->addBusManager( usbm0 );

    /*node specifications of first bridge*/
    node.class_id = 0x40;
    node.index = 1;
    node.version = 0;

    for ( i = 0; i < NUM_NODES; i++ ) {
        node.index = i+1;
        while ( ! (acc[i] = urbintf->findNode( &node )) &&
            (!control_c_invoked) )
            sleep(1);

        printf( "Found node %d!\n", i );
    }

    if (!control_c_invoked) {

        struct timespec remtime, waittime;

```

```
printf("Starting up...\n");

int loop_count = 0;

while ( ! control_c_invoked ) {

    for ( i = 0; i < NUM_READS; i++ ) {
        req = acc[i]->newRequest();
        req->read( i % 4+1, MAX_PAYLOAD, URB_REQ_PRESP_MASK );
        req->setCallback( read_callback );
        acc[i]->submitRequest( req );
    }

    for ( i = 0; i < NUM_WRITES; i++ ) {
        req = acc[i]->newRequest();
        req->write( i % 4+1, MAX_PAYLOAD, URB_REQ_PRESP_MASK );
        req->setCallback( write_callback );
        acc[i]->submitRequest( req );
    }

    waittime.tv_sec = 0;
    waittime.tv_nsec = WAIT_MS*1000000L;

    while ( nanosleep( &waittime, &remtime ) < 0
            && !control_c_invoked )
        waittime = remtime;
    }
}

delete urbintf;

printf("Exited main loop.\n");

} //end of main
```

B.4 Core Algorithm

The core algorithm of the driver is shown in Figure B.2. In this figure, flow of events involved in all threads, as well as interactions between the threads are shown. The blocked state of threads is shown by horizontal bars, and the blocked thread is signalled by the arrows that are drawn across the lanes. A lane is a separate region where the execution of a single thread is shown.

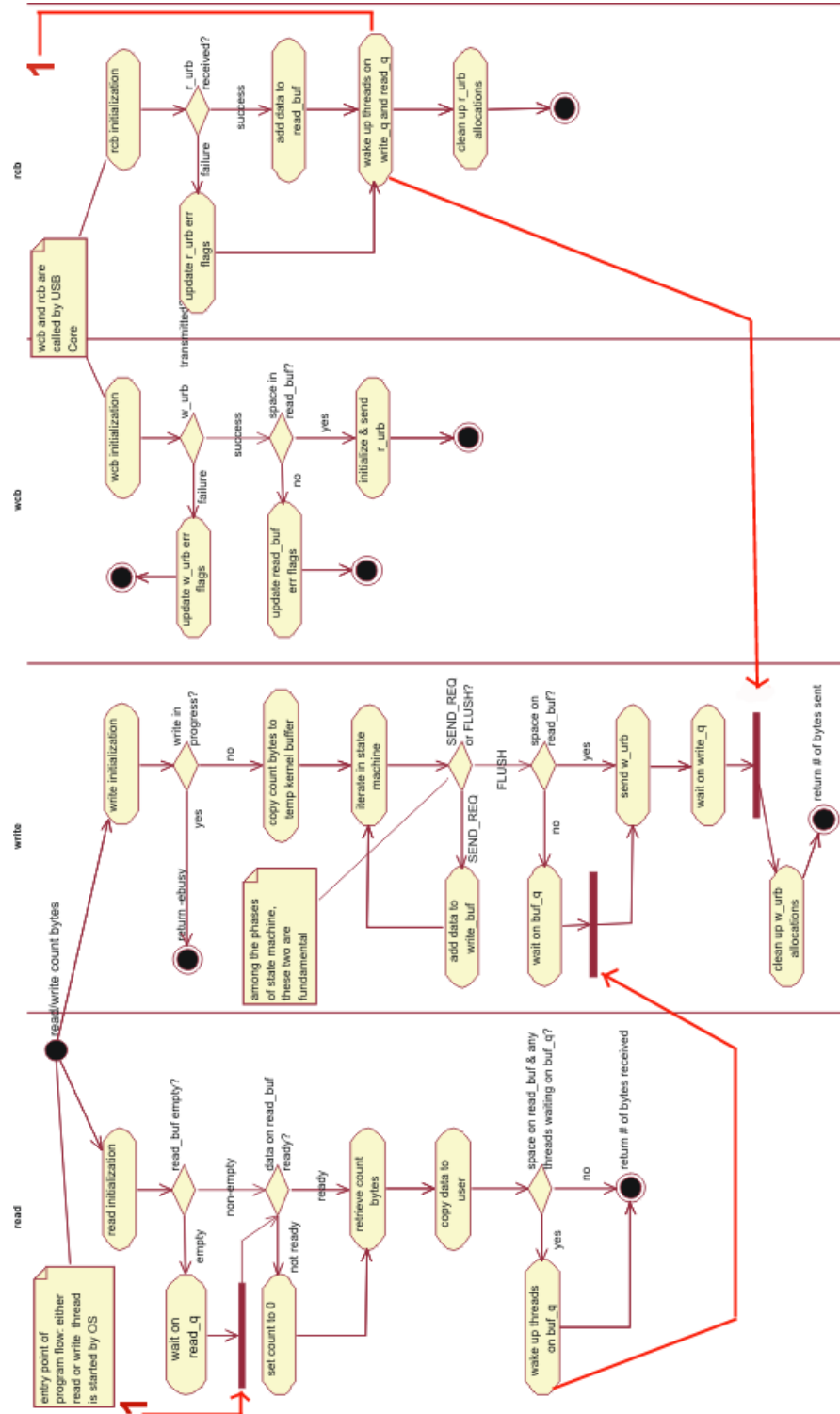


Figure B.2: Activity diagram showing flow of events involved in all threads. Interaction between the threads, signaling blocked threads, are indicated by arrows across the lanes.