THE UNIVERSAL ROBOT BUS: A LOCAL COMMUNICATION INFRASTRUCTURE FOR SMALL ROBOTS

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

Akın Avcı

December, 2008

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.

Dr. Cengiz Çelik

Approved for the Institute of Engineering and Science:

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

ABSTRACT

THE UNIVERSAL ROBOT BUS: A LOCAL COMMUNICATION INFRASTRUCTURE FOR SMALL ROBOTS

Akın Avcı
M.S. in Computer Engineering
Supervisor: Assist. Prof. Dr. Uluç Saranlı
December, 2008

Design and construction of small autonomous mobile robots is a challenging task that involves the selection, interfacing and programming of a large number of sensor and actuator components. Facilitating this tedious process requires modularity and extensibility in both hardware and software components. This thesis concerns the development of a real-time infrastructural architecture called the Universal Robot Bus (URB), based on the popular Inter-Integrated Circuit (I^2C) bus standard. The main purpose of the URB is the rapid development and real-time interfacing of local nodes controlling small sensor and actuator components distributed on a mobile robot platform. It is designed to be very lightweight and efficient, with real-time support for RS232 or USB connections to a central computer.

The URB infrastructure is inspired from the RiSEBus architecture, which is also an internal communication protocol for mobile robots, and developed to fit our requirements. URB offers a modular and extensible architecture for rapid and frequent changes to the platform design. Mobile robots also need to perform accurate sensory processing and estimation in order to operate in unstructured environments. Hence, the URB also supports real-time operations with reliable hardware and software components.

The first novel contribution of this thesis is the design and implementation of automatic synchronization of data acquisition across multiple nodes. Our synchronization algorithm ensures that each node completes data acquisition tasks simultaneously well before read operations. Our experiments also prove that each individual node acquires data at approximately the same time instant. The second major contribution of this thesis is the incorporation of automated and

unsupervised data acquisition across multiple nodes into the URB protocol. Autonomous data acquisition helps acquire periodic and frequently needed data over nodes. This enhancement reduces the computational load on the central processing unit and reduces bandwidth costs over the communication medium. This thesis also servers a survey on network architectures and protocols, network applications in robotics, synchronization algorithms, and applications of synchronization in robotics besides implementation details of the URB.

Keywords: Communication protocol, fieldbus, real-time communication, mobile robotics, distributed systems, clock synchronization, time synchronization.

ÖZET

EVRENSEL ROBOT VERİYOLU: KÜÇÜK ROBOTLAR İÇİN YEREL HABERLEŞME ALTYAPISI

Akın Avcı
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Assist. Prof. Dr. Uluç Saranlı
Aralık, 2008

Küçük özerk robotların tasarımı ve yapımı birçok algılayıcının seçilmesini, programlanmasını ve ortak bir arayüz geliştirilmesini de kapsayan zorlu bir iştir. Bu kapsamlı işlemi kolaylaştırmak, hem donanım hem de yazılım üzerinde modülerliği ve genişletilebilirliği sağlamayı gerektirir. Bu tez, I^2C tabanlı ve gerçek zamanlı altyapısal bir mimari olan Evrensel Robot Veriyolu (URB)'nun geliştirilmesiyle ilgilidir. URB'nin asıl amacı hareket eden bir robot platformu üzerinde dağınık halde bulunan, küçük algılayıcıları ve hareket unsurlarını kontrol eden yerel düğümlerin hızlı ve gerçek zamanlı olarak sisteme dahil edilmesidir. URB, merkezi bir bilgisayara gerçek zamanlı RS232 ve USB bağlantı desteği sağlayan etkili ve zahmetsiz bir altyapı olarak tasarlanmıştır.

URB altyapısı, benzer şekilde hareketli robotlarda dahili haberleşmeyi sağlayan RiSEBus'tan esinlenilerek geliştirilmiş olmakla birlikte, bizim ihtiyaçlarımıza göre iyileştirilmiştir. URB protokolü hızlı ve sık platform tasarım değişiklikleri için modüler ve genişletilebilir bir mimari önermektedir. Ayrıca, hareketli robotlar yapısal olarak düzensiz ortamlarda çalışabilmek için kesin algısal işlemlere ve tahminlere ihtiyaç duyarlar. Bu nedenle, URB güvenilir donanım ve yazılım bileşenleriyle gerçek zamanlı işlemleri desteklemektedir.

Bu tezin ilk önemli katkısı, çoklu düğümler üzerinden veri edinmenin otomatik olarak senkronize edilmesinin tasarımı ve uygulamasıdır. Senkronizasyon algoritmamız, veri okuma işlemlerinden önce her düğümün veri edinme işlemlerinin aynı anda bitirilmesini garanti etmektedir. Ayrıca deneylerimiz de her düğümün küçük farklarla aynı anda veri edindiklerini kanıtlamaktadır. Bu tezin ikinci katkısı ise, çoklu düğümler üzerinden otomatikleştirilmiş ve denetlenmemiş veri edinimidir. Özerk veri edinimi, periyodik olarak ve sıkça ihtiyaç duyulan verilerin düğümler

üzerinden edinilmesini sağlayan bir işlemdir. Bu geliştirme merkezi işlemci birimi üzerindeki hesaplama yükünü azaltırken, haberleşme ortamı üzerindeki bant genişliği masrafını da azaltır. Bu tez URB'nin uygulama detaylarının yanında ayrıca ağ mimarileri ve protokolleri, ağ uygulamalarının robot sistemlerindeki yeri, senkronizasyon algoritmaları ve robot sistemlerinde senkronizasyon uygulamaları üzerine bir araştırma olarak da hizmet etmektedir.

Anahtar sözcükler: Haberleşme protokolü, alan veriyolu, gerçek zamanlı haberleşme, hareketli robotlar, dağınık sistemler, saat senkronizasyonu, zaman senkronizasyonu.

Acknowledgement

I would like to thank my supervisor, Assistant Professor Dr. Uluç Saranlı for his guidance throughout my study. I met Dr. Saranlı when I was a senior student. He offered me an excellent opportunity, when I asked about studying with him as a master degree student . I feel lucky to have experienced the process of researching and developing consistent solutions to various research problems in the light of his guidance. His generous help and tremendous support over these three years carried me forward to this day. This work is an achievement of his continuous encouragement and invaluable advice. I also admire his enthusiasm for the scientific pursuit and endless energy.

I am very grateful to Assistant Professor Afşar Saranlı, Assistant Professor Yiğit Yazıcıoğlu and Professor Kemal Leblebicioğlu from Middle East Technical University (METU) for their help and brilliant ideas in Project RHex. I learned a lot from them during our research meetings and studies.

I am thankful to the members of the Bilkent Dexterous Robotics and Locomotion (BDRL) Group, Ömür Arslan, Sıtar Kortik, Cihan Öztürk, and Tuğba Yıldız.

I am also appreciative of the financial support I received through a fellowship from TÜBİTAK, the Scientific and Technical Research Council of Turkey.

Last, but never the least I would like to thank my parents, Yakup and Gülsüm Avcı, and beloved girlfriend İmran Akça for their love, support and encouragement.

To my family
&
my dear

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The objective of this thesis is to provide information about a communication infrastructure designed for internal communication within mobile robots. The thesis, therefore, presents real-time communication capabilities, protocol and properties of the network connecting sensors, actuators and processing units distributed over a robot.

1.1 Robotic Systems vs Embedded Systems

An embedded system is a special purpose computer dedicated to one or a few specific tasks. Embedded systems often contain processors that are not reprogrammable by the end-user and they should often satisfy real-time computing constraints. Some examples of embedded systems are microwave ovens' control panels, chips monitoring automobile functions like Anti-lock Brake System (ABS), digital watches, missile guidance systems, tanks' turret targeting systems, and traffic lights. As illustrated by these examples, embedded systems offer stable platforms closed to any modifications with limited user interface, sometimes none.

On the other hand, robots are machines capable of sensing environmental

changes and making decisions based on their own and environmental states. There are a variety of categories of robots such as assistive, industrial, humanoid and mobile robots. In contrast to embedded systems, robots should offer a re-programmable and modular structure. For instance, Programmable Universal Manipulation Arm (PUMA) is a robot arm that is used for a variety of industrial purposes. Behavior (software) of this robot may be changed according to type of usage and one might even change the end effector for different manipulation tasks.

Among other types of robots, mobile robots present further challenges. Mobile robots are strictly real-time platforms that are not fixed to one physical location. So, mobile robots can also be classified depending on the environment where they travel: land, underwater and aerial robots. Because of their nature, mobile robots should be concerned about energy, weight and space constraints as well. Any extension to a mobile robot would increase the energy cost, weight as well as the volume of the robot, effecting its dynamics and functionality.

1.2 Robotic System Architectures

Traditionally, a central architecture (Figure 1.1) is used to connect all sensors, actuators and other peripherals to a central processor within embedded systems and robots[30]. This simple architecture is easy to manage but it results in a large amount of cabling. For robots in particular, modularity and extensibility are two major constraints and large amounts of cabling makes it difficult to make any modifications.

Distributed architectures constitute a viable alternative to centralized architectures and are illustrated in Figure 1.2. This type of architecture has some advantages with respect to centralized architectures. First of all, there is a minimum amount of cabling since all the peripherals use the same communication medium. Secondly, it is easy to make changes over this architecture. Each functional block can be located anywhere on the whole system. This approach not

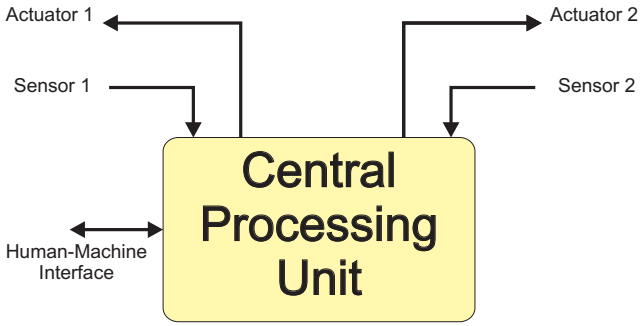

Figure 1.1: Traditional central architecture used mostly for simple embedded systems. All sensors, actuators and other peripherals are linked to a central processor.

only reduces the complexity of the system but also increases its modularity. Last of all, distributed architecture supports flexibility to extend the system without affecting functionalities of other peripherals.

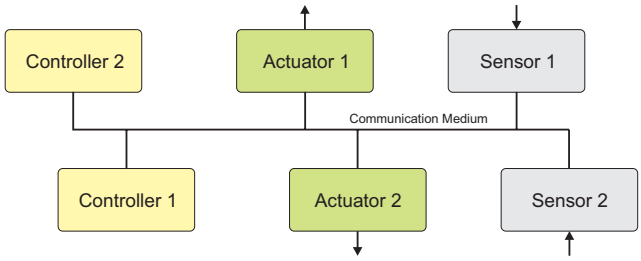

Figure 1.2: Distributed architecture with sensors, controllers and actuator are linked to the same communication medium.

As a result of the observations above, we can conclude that distributed architectures have more powerful characteristics than central architectures. However, distributed peripherals should be connected to each other over a communication medium which is sufficiently real-time and reliable enough.

1.3 Motivation

As already described in Section 1.1, robots are complex systems that should support modularity and extensibility. Any modifications would cause extensive changes over the software of the system and affect the hardware architecture as

well. In order to simplify modifications to the system, one could choose a distributed architecture that accepts extensions and modifications without effecting already installed components. This notion emphasizes the need for a communication medium that easily supports modifications.

The Universal Serial Bus (USB) is a good example of a modular communication infrastructure. USB is designed for connecting many different peripherals to personal computers and to allow adding and removing devices without rebooting the computer (*plug-and-play*). USB is a standardized and modular cable bus that supports data exchange between a host computer and a wide range of simultaneously accessible peripherals[1]. However, USB is not designed to satisfy real-time constraints and is not deterministic enough. These are essential components for most robotic systems. For this reason, it is not suitable to use robotic systems.

Communication mediums supporting real-time data transmission are called *fieldbuses*[32]. Today fieldbuses are used in many different industries such as automobile, building automation, and robotics. Our goal is to build a unique fieldbus applied to mobile robots with the constraints below:

- **Modularity and Extensibility:** Most mobile robots are designed for different field applications and these require rapid hardware and software changes. Modular robots are composed of modules that can be removed or re-connected. Modifications would yield systems with different functionalities for different tasks. Similarly, some tasks may need extensions to the system like adding a new sensor for detecting obstacles. A fieldbus should allow changes and adapt the system to modifications and extensions.
- **Real-time Performance:** Autonomous mobile robots operate in environments that are not predictable and can change dynamically. Mobile robots should decide and act rapidly to cope with these unpredictable and dynamic environments. Robots with decentralized architecture should also handle data transmission in a predictable and acceptable way so that they can act fast enough to environmental changes.
- **Reliabilty and Robustness:** Autonomous mobile robots operating in

dynamic environments in an unsupervised manner require a reliable electromechanical design with simplified cabling. Any mechanical or electrical failures would result in loss of functionality for the whole system. On the other hand, the internal communication medium should also minimize errors and should be able to recover from failures. As it can be observed, both hardware and software design are the key factors that effect the reliability of a mobile robot.

1.4 Structure of This Thesis

Chapter 2 introduces various network standards and communication mediums used for mobile robots. It also covers the details of previous researches about this subject. Chapter 3 explains the communication protocol, internal mechanism, and implementation details of URB. Chapter 4 and Section 5.1 covers the experiments and explains the performance criteria that is used to gauge the system. Finally Chapter 5 concludes with the explanation of the resulting system and operation environments of the system.

Chapter 2

Background

During the last decade, complexity of embedded and robotic systems increased substantially. This resulted in more complex hardware and software architectures. Consequently, researchers recognized the advantages of distributed architectures. As mentioned in Section 1.2, distributed architectures reduce cabling complexity but they bring about a need for inter-processor network protocols. One can find more than sixty *standard* communication protocols widely used in industrial applications[48].

This chapter overviews standard network models and some of the inter-processor communication protocols widely used for robotic and automation applications, as well as various applications of these protocols. It discusses the advantages and disadvantages of specified protocols and applications.

2.1 Network Standards

Open Systems Interconnection Basic Reference Model (shortly *OSI Model*) was first published in 1984 as an ISO standard (ISO 7498)[32]. In the early days of networking technologies, each developer had their own way of networking. This diversity causes many problems both in development and applications level. For

instance, there are many networking protocols currently used in personal computers and one wants to change the communication medium (LAN, WLAN, Dial-up Modem) without effecting or changing any software on the system. OSI model provides this flexibility by defining a standard for all communication protocols. The purpose of OSI Model is to coordinate development standards. OSI Model introduces a 7 layered model as represented in Figure 2.1. A layer is a collection of related functions that provides services to upper layers and get services from the lower layers. Each layer is described explicitly and problems in data exchange could be isolated in each layer. This model also gives the ability to use different standards for different layers without changing the entire system. All 7 layers of the OSI Reference Model are illustrated in Figure 2.1.

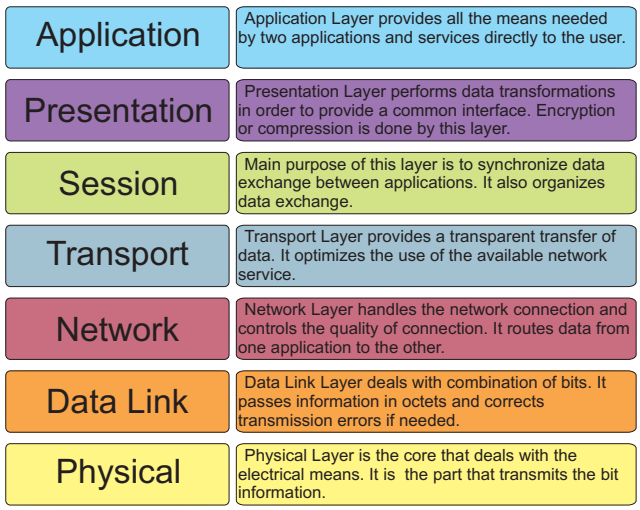

Figure 2.1: OSI Reference Model with explanation [32].

The OSI Reference Model was primarily developed for Local Area Network (LAN) and Wide Area Network (WAN) architectures[32]. A lot of distinctions between LAN/WAN architectures and fieldbus applications have been published but it is difficult to clearly distinguish these two. Fieldbuses are used for automation and control systems in order to communicate with sensors, actuators, and devices. Data sent over a fieldbus is usually less than 100 bytes and it is expected to be delivered to target within a second for real-time requirements. On the other hand, LAN/WAN systems are usually used for office, management, or visualization applications and do not need to support real-time transactions. In general, LAN/WAN systems are used to link computers to printers, computers,

or multimedia by transmitting data larger than 1 kbyte size without any time constraints.

Even though some of the fieldbus applications such as LON bus are introduced as a 7 layer OSI system, it is hard to fulfill strong real-time and short reaction time requirements with a 7 layer model. Furthermore, fieldbuses for robotic applications should satisfy hard real-time constraints by receiving data from external peripherals and sending commands from main processors to other peripherals . So in our design, we used a much more appropriate model for our domain that is also suggested for fieldbus applications. This model is called *Enhanced Performance Architecture (EPA)*. EPA implements only three layers of OSI Reference Model: application, data link and physical layers. This reduced model increases the performance of the network and reduces reaction time. Omitted layers might be implemented in these three layers if needed in order to not loose functionality provided by OSI Model. EPA also offers sufficient flexibility for a modular network protocol.

2.2 Types of Physical Media

Each network needs a physical medium to connect nodes together. Physical medium is the material substance which is used to transmit communication signals. There are several types of media used for linking nodes and we overview some of them in this section.

2.2.1 Twisted Pair

Twisted pair wires are composed of two insulated wires that are twisted around each other. Each twisted pair carries opposite and equal signals and these signals are added on the receiver side to obtain the the exact signal. By this way, twisted pair cables minimize electromagnetic interference from external sources. There are three major types of twisted pair wires:

2.2.1.1 Shielded Twisted Pair

Shielded Twisted Pair (*STP*) cabling includes shielding over each twisted pair and a shield overall the twisted pairs as illustrated in Figure 2.2.b. A shield is a conductor used for eliminating external electromagnetic interference and *crosstalk* between pairs. STP is mostly used for industrial environments exposed to high electromagnetic field up to a frequency of 155Mbps.

2.2.1.2 Unshielded Twisted Pair

Unshielded Twisted Pair (*UTP*) is another way of twisted pair cabling but this type of twisted pair wiring doesn't include any shields as displayed in Figure 2.2.a. UTP based communication media can transmit data up to 100Mbps. Frequencies greater than this value don't allow reliable transmission.

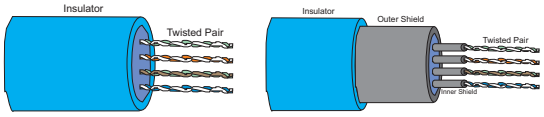

Figure 2.2: Illustrations of (a)Unshielded Twisted Pair (b)Shielded Twisted Pair

2.2.2 Coaxial Cable

Caoxial cable is a communication medium designed for reliable transmission of radio frequencies. It is composed of a solid inner wire in the core and outer plaited layer of wire. Coaxial cabling is illustrated in Figure 2.3. The term coaxial cames from the inner and outer wires sharing the same axis.

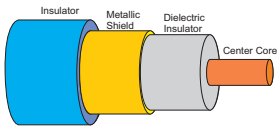

Figure 2.3: Coaxial Cable Structure

Structure of coaxial cables provides protection from external electromagnetic fields. Electromagnetic field only occurs between inner and outer conductors.

Coaxial cables can support up to 10 Mbps data transmission.

2.2.3 Optical Fiber

Fiber optic is a glass or plastic fiber that carries light along its length. Since fibers carry light instead of electrons, fiber optics permits data transmission over longer distances with higher bandwidths. Data rates depend on the equipment that lights the fiber. Structure of a fiber is illustrated in Figure 2.4.

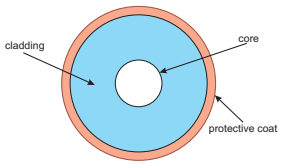

Figure 2.4: Optical Fiber Structure

Optical fibers are not affected by electromagnetic and radio frequency interferences since fibers uses light for data transmission and light is not affected by these interferences. This situation makes optical fibers suitable for almost all environmental conditions.

2.2.4 Radio Frequency (RF)

Radio Frequency (*RF*) denotes frequencies of radio waves ranging from 3 Hz to 300 GHz. RF signals are radiated by an antenna with alternating current flow over the antenna. RF signals are widely used for several different wireless communication applications like TV broadcasting and mobile communications.

RF signals are categorized depending on their frequencies. Signals ranging from 300MHz to 300 GHz are called microwave and microwave signals are used for Bluetooth, wireless LAN and satellite communications. Nightvision, remote control and Infrared Data Access (*IrDA*) applications use infrared signals that have a frequency range of 3 GHz to 400 THz.

2.3 Applications of Network Technologies in Robotics

In section 2.1, we defined OSI and EPA models and explained differences between the two models. There are numerous applications based on the two network models and there are also various network applications in robotics. This section gives a brief explanation for the usage of network applications in robotics.

2.3.1 Teleoperation

Robots can be roughly categorized into two groups based on their decision making mechanism. One of these categories is autonomous robots and the other is human-assisted robots. Autonomous robots are robots that perform desired and pre-defined tasks without continous human assistance. On the other hand human-assisted robots need someone to control the robot and make decisions on behalf of the robot itself.

Human-assisted robots are driven by humans so there has to be a communication link between the human and the robot. The communication between human and robot is called *teleoperation*. Teleoperation is the operation of a machine from a distance. In other words it means 'remote control' of robots. Invention of radio communication underlie the history of teleoperations and WLAN, Wi-Fi, Bluetooth and Deep Space Network (*DSN*) are some protocols used for teleoperation of robots.

2.3.2 Swarm Robotics

As opposed to single robot applications, swarm robotics concerns the coordination of multiple robots. Swarm robotics evolved after research on artificial swarm intelligence and biological studies of insects. Swarm robots are relatively simple with respect to single robots but each robot in a swarm shares calculations and

environmental interactions with other robots in the swarm.

Sharing data between robots in a swarm necessitates the usage of a communication medium between robots. Zigbee, CAN, Bluetooth and Wi-Fi are some of the examples to network protocols used for swarm communication.

2.3.3 Internal Communication Infrastructures

Industrial network systems for real-time distributed control are called fieldbuses [32]. Based on this definition, the most important characteristic of fieldbuses is the real-time data transmission. Fieldbus systems have a wide range of applicability from automation to robotic applications because of real-time constraints.

In automation applications, fieldbus is a communication medium that carries data from a distributed system of sensors, actuators, lights and switches to the main controller of the system. This time critical data is used for ensuring the stability of the system. On the other hand, fieldbuses are also used as the internal communication infrastructure for robots that adopt distributed architectures. In such a system, sensors, actuators and controllers are distributed over the robot as illustrated in Figure 1.2.

There are numerous fieldbus protocols used for internal communication within robots. Robots, especially mobile robots, that need hard real-time operating constraints mostly use a fieldbus architecture. In this thesis, we are only concerned with network protocols used for internal communication within robots and the following sections introduce some of the mostly used real-time network protocols as internal communication infrastructures.

2.4 Physical Connectivity Alternatives

In order to share data between elements of a network, each device on the network must be physically connected. This section investigates some of the alternatives

for bus standards to interface devices.

2.4.1 Universal Serial Bus (USB)

The Universal Serial Bus (*USB*) specification was first introduced in 1995 and promoted by Intel. The main motivation of USB designers was to build a flexible and low cost protocol with high transfer rates that support real-time data transmission for audio and video.

USB physical interconnect is a tiered star topology that can be extended up to 7 tiers [1]. The USB *hub*, that connects each device to the USB and other hubs, and is the center of each star. Root hub for the USB is the center of the entire architecture and is called USB *host*.

USB is a polled bus and every transfer is initiated by an interface called the USB *host controller*. Each transmission is initiated with a starting packet, called *token packet*, transmitted by the host controller. This packet includes the address and end point information for the desired device. Connection between the host controller and linked devices is established following the reception of this packet. Once a connection is established, the host controller either sends or receives data packets. In order to achieve a reliable communication, response to data packets are *handshake* packets indicating that the transmission was successful.

Besides being a reliable and fast data transmission medium, USB is also a flexible bus. USB implements three operating modes for various tasks. In order to link interactive devices like mice, keyboards or game peripherals, the USB host controller supports *low speed* connections. Tasks with real-time constraints may need faster communication so USB implements *full speed* and *high speed* operating modes. Full speed mode is used for audio peripherals like microphones and speakers and provides a data rate of 12 Mb/s. Moreover, the high speed mode can used for real-time tasks that need high data transmission. Some examples for high speed operating mode might be video and storage tasks. Furthermore, the USB protocol enables auto-detection of newly connected peripherals. Each

device is connected over a hub and each hub has a status bit for connected devices. The host controller queries for these bits in order to detect a new connection or disconnection.

2.4.2 Firewire (IEEE-1394)

Firewire is a communication interface targeting personal peripheral communication market like USB. It was developed by Apple Computer in 1995 for high speed communications and standardized as IEEE-1394 ("A High Performance Serial Backplane Bus"). Firewire is designed up to the transport layer and it provides asynchronous and isochronous real-time communication for multimedia devices. It implements a bitwise arbitration algorithm for medium access control. However, firewire does not require configuration at startup and supports hot-plugging like USB. It supports communication speeds ranging from 12.5 Mbit/s to 50 Mbit/s[6].

2.4.3 Recommended Standard 232 (RS232)

RS232 is an asynchronous serial communication interface defined by the Electric Industries Association in 1962 as a recommended standard (RS). RS232 is also a *complete standard*[42] and in a complete standard voltage and signal levels, pin configurations are all defined and it needs minimal amount of control information. Furthermore, the Universal Asynchronous Receiver/Transmitter (*UART*) is the primary component of the serial communication systems. UART takes bytes of data and transmits each bit over a serial bus in a sequential order. On the receiver side, UART receives data bits and assembles exact data in bytes.

RS232 operates in a wide range of voltage values. Different from most of the communications protocols, RS232 defines positive voltage as +15V and negative voltage as -15V. However it also supports +5V to +15V so RS232 can work with various systems. This flexibility also brings some limitations to cable length and operation frequency. The oscillation of voltage levels from +15V to -15V causes

electromagnetic interference and limits the baud rate of the bus and cable length. Baud rate can be defined as the bits transmitted within a second and RS232 operates in a range of 1200 to 115200 baud rate. Besides, different baud rate values have different limitations over cabling. 2400 baud transmission permits 30 meters of cabling while 19200 baud transmission permits a maximum of 6 meters cabling.

2.4.4 Peripheral Component Interconnect (PCI)

Peripheral Component Interconnect or **PCI Standard** was first designed by Intel Architecture Lab in 1990 in order to build a low cost, flexible, high performance local bus[21]. The motivation behind the design of PCI bus was to connect components to a computer in order to increase interchangeability.

PCI presents a multi-master and peer-to-peer architecture. Each component linked to the bus is able to be the master of the bus and transmit data directly to other peripherals. PCI devices are plug and play devices so when a new peripheral is linked to the bus, the system triggers a configuration task to use the new peripheral without external intervention.

PCI buses can operate in 32 bit data paths or 64 bit data paths and 33 MHz bus clock frequencies or 66 MHz bus clock frequencies. As a result of different configurations, PCI can transmit data at a rate of 1Gbps for a 33MHz system clock and 32 bit data path. PCI also can work at a rate of 4Gbps for a 66 MHz clock speed and 64 bit data path. Resulting performance is acceptable for real-time applications with guaranteed low latency.

2.4.5 Controller Area Network (CAN)

Until early 80s, point-to-point wiring systems were used for automobile technology. But this implementation made hardware systems more complex and any

new technology added to the vehicles results in increased cabling cost. New extensions and modifications affected the design of automobiles. In order to solve this problem, Bosch GmbH introduced CAN bus in 1988[37].

CAN is a serial communication infrastructure especially designed for small scale networking purposes in decentralized architectures. CAN takes a dynamic approach, using a priority based algorithm to decide which node is allowed to send a message. Contrary to dynamic approaches, static approach uses a fixed time interval for each connected station to send their data as described in Section 2.5.2. In the dynamic approach, each node acts like a master to the bus and the one with the highest priority takes the control of the bus. After taking control of the bus, sender broadcasts the message and related nodes receive the message. This approach is good for transmitting most urgent messages but it does not guarantee delivery of less urgent messages [45].

CAN applications range from automotive applications to robotic applications, because of its very efficient real-time performance and reliability[36]. Since CAN automatically checks errors and re-transmits data, it doesn't require extra operations for checking errors.

2.4.6 Inter-Integrated Circuit (I^2C)

I^2C Bus was first invented by Philips in order to connect low speed devices by using two bi-directional lines[35]. These two lines are called *Serial Data* (SDA) and *Serial Clock* (SCL), which are both *open-drain* lines. *Open-drain* is an interfering technique widely used for linking multiple devices on a single line. Technically, *open-drain* devices are actively sinking current (logic 0) or are high impedance, but they do not actively feed the circuit with current. Both SDA and SCL lines must be pulled-up with resistors.

I^2C Bus protocol allows multiple master applications. This means that each device connected to the bus can initiate a transmission over the bus. It can also be implemented as a single master bus. In URB we used I^2C as a single master

device as displayed in Figure 2.5.

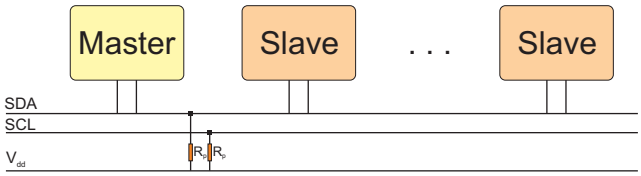

Figure 2.5: Single master application of I^2C as it is used in URB.

The reference design of I^2C Bus has 7 bit address space with 16 reserved addresses, so a total of 112 devices can be linked over a single I^2C Bus[35]. I^2C has 5 different operation modes: *Low-Speed Mode*(10kbit/s), *Standard Mode*(100kbit/s), *Fast Mode*(400kbit/s), *Fast Mode Plus*(1Mbit/s), and *High Speed Mode*(3.4Mbit/s).

2.5 Network Protocols Relevant to Robot Communications

There are many communication protocols used as internal communication infrastructure for robots. Within these protocols, one needs to choose the most suitable protocol for a specific application. All embedded and robotic applications have different kinds of constraints and requirements. Design changes depending on the specific functionality of the bus, would increase the performance of the whole system. In this section, we discuss some of the widely used communication protocols for fieldbus applications.

2.5.1 Local Operating Network (LON)

Local Operating Network (**LON**) is a computer network designed for small geographic areas like home, office or group of buildings. LON is a 7 layered, packet based, peer-to-peer(P2P) network which also supports authentication and priority based messaging[11]. LON uses a diverse connectivity between participant nodes and this makes LON work as a peer-to-peer network. Echelon Corp. and

their partner Toshiba developed a sophisticated microcontroller with networking and interfacing named Neuron. These chips are composed of 2 controllers for networking purposes and one for executing user programs[47].

LON is designed for control systems and most control systems have common requirements that are independent of application [11]. Control networks support frequent, reliable, secure communication, short message formats and P2P functionality. In order to satisfy the needs of control systems, LON is designed to be flexible and scalable. LON systems runs on various media like twisted pair wire, radio and power line. This property gives developers the flexibility of building the system on a suitable media and port whole system to another media like power line in places where the cabling cost is very high. This is handled by building the physical connection of nodes through a transceiver[47].

LON systems are based around a protocol called **LonTalk** and this protocol works like the Ethernet protocol. As in Ethernet protocols, simultaneous transmission requests are possible. Ethernet handles collisions by using the Medium Access Control (MAC) algorithm but this does not eliminate collisions. If a collision occurs, the MAC algorithm defines a random delay for each peripheral and then re-transmits data. This situation is not suitable for control networks since the MAC algorithm causes non-determinism and does not guarantee data transmission. So, LON implements a different MAC algorithm called *predictive p-persistent CSMA* protocol. This protocol enhancement reduces collisions over the connection medium but cannot eliminate all collisions.

LonTalk protocol supports four addressing types[11]. *Physical addressing* uses a 48 bit node identifier which is embedded in the hardware[17]. *Device addressing* handles addressing in a more effective manner by categorizing the given address according to the groups, subnets. etc. LonTalk also supports *group addressing* so that a message can be delivered to a group of devices. In order to send a message to the whole system, LonTalk implements *broadcast addressing*. LonTalk can support up to *32385* devices and *256* groups.

2.5.2 Time-Triggered Protocol (TTP)

Communication networks are categorized into two basic classes[25]: *event-triggered (ET)* and *time-triggered (TT)* architectures. ET network architectures initiate data transmission with the occurrence of specific events, such interrupts. Each task and communication events are triggered by interrupts. Because of this, replica determinism and collision problems cannot be solved[23]. TT protocols handles this problem by initiating events of each node depending on the progression of the *global time*.

Time-Triggered Protocol (**TTP**) is based on Newtonian physics[24]. A time interval from current time instant to an instant in the future consists of infinite number of time instants. So TTP samples this interval and triggers events with respect to the sample time instants. This approach introduces a new concept in communication networks: *synchronization*. Each node connected to system has its own clock and clocks of all nodes should be synchronized for accurate timing of events. So, TTP implements a synchronization algorithm in order to acquire a global time base with an acceptable error.

TTP is designed to be a fault-tolerant real-time protocol with guaranteed timeliness. Each node is replicated[41] or grouped so that any failure in one of the nodes can be recovered by a duplicate of defective node. Replicated nodes perform the same operations and stores the exact state of the running node. TTP also uses two bidirectional channels as communication medium in order to tolerate communication errors.

2.5.3 Attached Resource Computer Network (ARCNET)

Attached Resource Computer Network (**ARCNET**) is a real time protocol designed by John Murphy at Datapoint Corp. in 1976. ARCNET was designed for fast transfer of large amounts of data over various types of media like coax, twisted pair or optical waveguides[3]. It is intended for office applications but is also an ideal fieldbus with time predictable message delivery[10].

ARCNET uses a token passing algorithm as its Medium Access Control mechanism. A token is passed through participants of the bus and the one owning the token has control of the network. Each participant can transfer up to 508 bytes of data at a time and should then pass the token to another participant. The token passing algorithm is deterministic where each participant has a predetermined time for controlling the network. However this means extra work for each participant and it becomes difficult to administer the bus.

ARCNET also supports handshaking and checksum calculation for reliability of the data and automatic integration of participants defined by a unique ID range from 1 to 255. A new participant is detected by the neighboring participant and integrated to the network. Likewise, if any of the participants leave the network, the neighboring participant detects this and reconfigures the Next ID (NID) value stored for the token passing algorithm.

2.5.4 Building Automation and Control Network (BACNet)

In the late 80s, work on intelligent buildings and control networks caused the development of many networking protocols. One of the mostly used protocol for building automation is the Building Automation and Control Network (**BACNet**). BACNet was introduced in 1991 and became an American Society of Heating, Refrigerating and Air-conditioning Engineers (ASHRAE) standard in 1995.

BACNet eliminates 3 layers of the standard OSI model and implements only the application, network, datalink and physical layers. This way, some of the unnecessary operations are also eliminated. BACNet also supports confirmed and unconfirmed message transactions, as well as single or multiple options for networking technology[9].

BACNET implements an object based data representation in order to abstract the internal design of devices and prevent application dependency. There are 18

different object types such as *device*, *command*, *group*, etc. One device might contain one or many of these objects except the *device* object. Any device should have exactly one device object defining device information (like vendor name, model name, protocol version, etc.). Each of these objects has a unique 4 bytes object identifier defining object type and object instance.

2.6 Application Protocols Relevant to Internal Communications Within Robots

Section 2.5 discussed some of the mostly used network protocols. Today these protocols became standards and they are widely used for industrial, office, building automation, and control applications. But some of these protocols stand out with their convenience for mobile robotic and real-time applications.

CAN protocol is one of the mostly used protocols for robotic and real-time systems. Wargui and Rachid[53] proposed a decentralized architecture for mobile robots based on CAN. Decentralized architectures provide autonomy to each specialized unit. This means local control tasks are executed locally, reducing communication delays in local control loops.

Wargui and Rachid categorize network models into two: *probabilistic* and *deterministic* models. In the probabilistic model, all devices compete for access to the bus and in case of a collision, all devices back-off and try again after a random delay. Since this model doesn't guarantee exact timing and even delivery of message, this model is not suitable for real-time applications. On the other hand, deterministic models guarantee a predictable delay by using a "right to transmit". Satisfying real-time constraints with a deterministic model, they also mention the importance of modularity for robotic applications. A network protocol used for robotic systems should be easily extensible and reconfigurable for future device additions.

Fernandez and Souto[14] also proposed a USB like communication system

to share sensor and actuator data based on the CAN protocol. Different from [53], they developed a centralized system in order to detect new additions. The proposed system is capable of auto-detection of new modules and the master of the system loads appropriate drivers for the specified module.

Some researchers don't find the pure CAN protocol suitable for real-time applications and developed extensions to it. Perez and Posadas[34] presented a hybrid communication protocol between TTP and ETP based on CAN protocol. Furthermore, Hong and Kim[18] offer a bandwidth allocation algorithm to efficiently control the traffic in CAN. In pure CAN applications, real-time data might be interfered by non-real-time data transmissions. Real-time data consists of feedback control data and event data that should be delivered with an acceptable delay. Bandwidth allocation algorithm is used to satisfy the delay requirements for control and event based data by implementing a window scheduling algorithm.

Alike [18], Mock and Nett[29] proposed an enhancement for a Time-division Multiple-Access (TDMA) protocol intended for real-time communications in robots. In pure TDMA, time-slots are allocated for the worst case and most of the time these slots are not even used. They solved this problem by sharing a time slot between events. This concept is designed for multiple access buses like WLAN and field-buses. Main idea is to give a higher priority to the to real-time messages and sharing the time slot with non-real-time message. By this way, this approach guarantees a predictable delay for real-time transmissions and optimizes the load over the bus.

The LON protocol is also used widely for robotic and real-time applications. Janet and Wiseman[22] approached the concept of real-time distributed systems by using the LON protocol. This study is actually a summary for the usage of the LON in three different types of projects: biped walking robot, hexapod colony, and complex autonomous robot. They denote the reason for choosing the LON for these projects is its flexibility. By using LON, the system can easily be expanded without rewiring or re-programming. Besides, the LON protocol supports both predefined timely events (*static*) and interrupt based events (*dynamic*).

All these studies confirm the need for a reliable, real-time, and deterministic

network protocol used in a distributed fashion. On the other hand, another problem arises within distributed architectures. Each device connected to the communication medium has its own time base and other devices don't know when the data is obtained. This problem is named node synchronization.

2.7 Node Synchronization and Determinacy

Time critical applications require a real-time clock to trigger events and detect exact time of occurance for an event[16]. In a central architecture, internal clock and timers are responsible for triggering and detecting timely events. However, in a distributed architecture, each peripheral has its own clock and there is no global time concept. Even if all the processors are started at the same time, clocks on processors may drift one second in every ten days[40]. Therefore, it is mandatory for clocks to be synchronized in a distributed architecture for accurate and deterministic timing of events. Generally, a virtual clock is generated locally for each peripheral depending on the global time base.

Synchronization of data is one of the main real-time constraints for a mobile robot platform capable of dynamic locomotion. Accuracy of the global timing of events directly affects the performance of the computational infrastructure. Sensory data received from external peripherals should be consistent and actuator commands should be accurately timed. Accurate synchronization of peripherals improve the performance of state estimations and environmental perception for highly mobile robots.

2.7.1 Types of Synchronization Algorithms

Clock synchronization within a distributed architecture requires a synchronization algorithm. A wide variety of algorithms have been proposed in the literature and these algorithms can be categorized under three classes[40]:

2.7.1.1 Convergence Based Algorithms

Convergence based synchronization algorithms are used to combine the values of all the processor clocks over a distributed system[40]. A function is used on each peripheral in order to calculate the drift of this associated processor, and the result of this function is a global time base for each processor. This function is called the *convergence function* and its primary goal is to minimize maximum deviation between clocks[15]. This function takes *N+1* arguments for *N* processors and first argument is used to define the processor evaluating the function.

Algorithms based on convergence functions implement a simple approach. Each peripheral reads the clock values of other peripherals at the end of each synchronization round. The convergence function is used to calculate the global time and the resulting time value is used to adjust the clock of each peripheral for the next synchronization round. Thereafter, each peripheral agrees on the same global time base and each of them converges to the common global time value.

2.7.1.2 Agreement Based Algorithms

Agreement based algorithms are another way of synchronizing clocks over a distributed architecture. The idea to synchronize processors' clocks is to sending messages between each processor and reaching an agreement for the global clock. In the absence of faulty processors, it is easy to reach an agreement[33]. But in most of the cases there might be one or more faulty clock sources.

Agreement based algorithms allow processors over a distributed system to agree on an action or set of values[40]. Agreement protocols start the synchronization process by broadcasting a message at a pre-agreed time instant[50]. After some message exchanges, it is the responsibility of an agreement based algorithm to define a global time value by using each received local clock value. It is also the algorithm's responsibility to define the agreement protocol. For instance, median, or modulus operations can be easily used to calculate the global time. Furthermore, faulty processor clock readings might cause a bad calculation and

they might be eliminated by the agreement algorithm.

2.7.1.3 Diffusion Based Algorithms

The main goal of diffusion based synchronization algorithms is to use local operations to achieve global agreement[27]. Different from agreement based algorithms, diffusion based algorithms force each peripheral to exchange and update information locally with their neighbors. Diffusion based algorithms are widely used for various applications like load-balancing[44], sensor networks and clock synchronization. The principle is to exchange data with neighboring peripherals and diffuse the global value to an average or highest value or lowest value. After a series of rounds, each peripheral would agree on a global value.

Diffusion based algorithms can be used for clock synchronization over a distributed system. Neighboring nodes exchange clock values and the average of values are used for synchronization[27]. Converging to highest and lowest values may lead erroneous situations if a node results a faulty clock value that is too low or too high, so it might mess up the synchronization.

2.7.2 Applications of Clock Synchronization

Section 2.7.1 gives the most common algorithms used for clock synchronization over distributed systems. These generic algorithms are used in various network applications and there are many applications of these algorithms. These applications can also be categorized under four basic classes:

2.7.2.1 Traditional Clock Synchronization Protocols

Most traditional clock synchronization protocols share the same basic design principles:

- Connectionless messaging protocol

- Exchange of clock information between server and clients/nodes
- Method to reduce effects of non-deterministic communication delay
- Update client time-stamp based on server clock

Network Time Protocol (*NTP*) is one of the oldest synchronization protocols. It is designed by David Mills and still in use since 1985. NTP is designed to support accurate, reliable time for financial, legal transactions, transportations and distributed systems and it is used in the Internet to synchronize clocks and coordinate time distribution[28]. The main idea of NTP is to generate a global time on a global server and each subnet connected to this server is synchronized according to the disseminated time-stamp.

Gergeleit proposed a clock synchronization algorithm specifically for the CAN bus[16]. Proposed synchronization algorithm is based on a master-slave configuration and it can be seen as an implementation of *a posteriori agreement* approach. According to this algorithm master of the CAN bus broadcasts a periodic start message that triggers a virtual clock on each slave. If the master is healthy and running properly, CAN guarantees delivery of broadcast messages and precision of virtual clocks.

With respect to real-time properties of distributed systems, the concept of temporal firewalls is also an interesting and useful one, presenting a mechanism to preserve real-time properties in the presence of independently operating subsystems[26]. Temporal firewall is a unidirectional control-free interface that connects the almost autonomous partitions of a system[26]. [34] proposes a temporal firewall approach used to synchronize subsystems linked to CAN bus.

2.7.2.2 Wireless Clock Synchronization Protocols

There are a few synchronization algorithms for wireless networks. Römer [39] also proposed one of these algorithms for ad hoc networks. Ad hoc networks are wireless mobile networks and characteristics of ad hoc networks makes traditional

synchronization protocols difficult to use. Due to limited connectivity range, each node is connected to only some of the nodes located in the range of communication and can communicate other nodes outside the range by using reachable devices as bridges. Hence, it is difficult to synchronize clocks of each device.

Römer proposed an algorithm to synchronize to generate time-stamps and use time-stamps for unsynchronized devices rather than synchronizing clocks of processors. As a message moves from hop to hop, each hop converts time-stamp of the message to its own time-stamp with some error. Error in time-stamp increases with the number of hops.

2.7.2.3 Receiver-Receiver Synchronization

Some of the clock synchronization algorithms show big differences from traditional approaches. Their ideas are to synchronize a set of receivers among themselves. Verissimo and Rodrigues [51] were the first to use this idea.

After Verissimo, PalChaudhuri[31] proposed a *Receiver-Receiver* clock synchronization algorithm for sensor networks. Clock synchronization is also important for sensor networks for data integration in sensors and sensor reading fusion. But the characteristics of sensor networks complicates the use of a traditional approach for sensor networks. Non-deterministic random time delay for a transmission of the synchronization signal, makes it difficult to make precise calculations.

According to Receiver-Receiver synchronization approach, a sender broadcasts a signal over the network and it is assumed that all receivers receive the synchronization signal. Each receiver on the network marks the time that they received the signal and exchange the information of reception time. Therefore, they approximately estimate the clock value of neighboring receivers and synchronize among all receivers.

2.7.2.4 Probabilistic Clock Synchronization

Arvind [5] proposed a probabilistic synchronization algorithm for a master-slave architecture. This synchronization protocol is composed of two main parts. First part is called Time Transmission Protocol (TTP) where the master sends its clock to slaves and each slave synchronize to master node clock. Second part is called Probabilistic Clock Synchronization (PCS) where slaves receiving the clock value of master estimate master clock compute the difference between its own clock and master's clock probabilistically.

The synchronization procedure is repeated every interval of time and precision of synchronization can be increased by increasing running frequency of the procedure. Arvind showed minimum number of messages needed for achieving a given maximum error.

2.7.3 Phase Locked Loops

There is also an old but well-known method used for synchronizing clocks and signals, called Phase-Lock Loop (PLL). PLL achieves synchronization of the frequency of a signal, with the frequency of a reference signal by using the phase difference between the two signals [13]. Phase-locked loops are widely used in radio, telecommunications, computers and other electronic applications since it has been invented.

2.7.3.1 History

Concept of synchronization dates back to 1920s[52]. Early studies on synchronization is to synchronize oscillator signals. Vincent[52] and Appleton[2] experimented and analyzed practical synchronization of oscillators. In the following 10 years, the synchronization problem became more interesting with the developments in communications. In 1932, a French engineer called De Bellescize[7] implemented first analog PLL circuit. Bellescize is known as the inventor of

'*coherent communications*' after his work on PLL circuits.

After Bellescize research on synchronization focused on synchronization of local oscillator in FM demodulation[20]. Travis[46] designed automatic frequency tuning for FM receivers. Until then, FM receivers are tuned manually and mechanically by the user and visual indicators were provided. Travis' design is composed of two primary elements: a tunable oscillator and a frequency discriminator used to tune the oscillator.

Advancements in synchronization of local oscillators lead to of the greatest inventions of the century: television. First versions of television are colorless and PLL is used to synchronize the image on the screen. PLL circuit keeps heads at the top of the screen and feet at the bottom of the screen[8]. Color televisions wouldn't also be possible without PLL. PLL circuit makes sure green remains green and red remains red[8].

First PLL integrated circuits (IC) appeared in 1965 and they were purely analog. With the development of cheap PLL ICs, PLL is started to be used widely in industry. In 1970s, PLL circuits drifted to digital territory and played a key role for digital communication devices.

2.7.3.2 Classification of PLL

First implementations of PLL were purely analog and mostly used for synchronizing clocks for radio communication. Purely analog versions of PLL is also known as "linear PLLs", illustrated in Figure 2.6. Linear PLLs are composed of three primary elements: phase detector, loop filter and voltage controlled oscillator (VOC). Phase and frequency differences of a reference signal and output signal are detected by the phase detector and according to this difference VOC generates an output signal with the same frequency and phase of the reference signal.

In the 1970s, the need for synchronizing digital signals motivated the digital implementations of PLLs. Although they were used for synchronizing digital

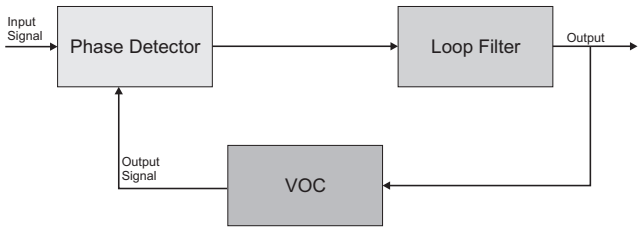

Figure 2.6: Block Diagram of the Phase Locked Loop

signals, first digital PLL systems were hybrid, also including analog peripherals. After a few years, first all-digital PLLs occurred and they are still widely used in ethernet, WLAN, digital signal processors (DSP), etc [19]. The idea behind digital PLLs was simply to take the difference between the reference signal and output signal. Result of this operation is also a digital signal and this signal would be used to adjust the output signal. This advancement proved the possibility of implementing a software PLL. Software PLLs provides the freedom for implementation but performance depends on a fast algorithm and effective implementation.

Chapter 3

URB: The Universal Robot Bus

We observed in previous chapters that extensibility and modularity are essential features for mobile robot designs as well as space optimization and power utilization. Considering these requirements on reliability, modularity, extensibility and computational power, we designed a distributed communication infrastructure. The proposed architecture provides local computational entities linked to a multi-task central processing unit. This central processing unit is responsible from coordinating all entities linked to the system so that they perform sensory acquisition and actuation tasks. This local communication architecture for robots is called The Universal Robot Bus (URB). This chapter describes the design details, concepts, protocol details, and architecture of the URB.

3.1 Overview of the URB Infrastructure

As noted above, a physically centralized control system where all signals physically connect to a single multi-task controller would be inappropriate. However, commanding all the necessary sensor and actuator devices from a central controller is a desirable method for simplicity and ease of development. For this purpose URB implements a *logically centralized* system architecture as illustrated in Figure 3.1. In this logical architecture, all sensor and actuator nodes work

together with the central processing unit (CPU) in order to accomplish required tasks. URB implements user level libraries both for the CPU and nodes, abstracting away from physical implementation details.

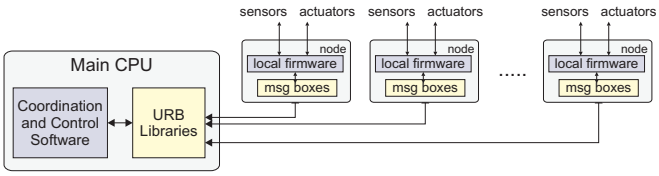

Figure 3.1: The Logical Topology of the URB System

The advantage of the URB architecture lies in its physical topology, illustrated in Figure 3.1. The physical topology of the URB system includes "bridges" that link the main CPU to nodes as illustrated in Figure 3.2. Since there might be multiple I^2C buses linked to the main CPU, bridges are essential to direct commands and requests sent to nodes. Bridges are transparent to programmers and are similar to USB host controllers. However, USB host controllers are different from URB bridges in their hardware topology. URB implements a two tiered architecture linked with bridges while USB implements a multiple tiered system separated with hubs.

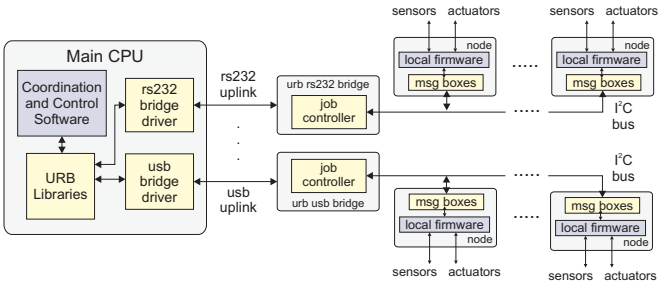

Figure 3.2: The Physical Topology of the URB System

As illustrated in Figure 3.2, the physical topology of the URB is composed of three main components, described in the following sections.

3.1.1 The URB Central Processing Unit (CPU)

URB CPU is the central authority of the bus and is responsible from the coordination of all devices linked to the bus. It collects sensory data, sends actuator

actions and implements complex control and decision algorithms. It is usually an embedded PC or a similarly powerful computational device.

URB CPU also implements drivers that handle communication details and convenient Application Programming Interface (API). Different types of bridge implementations require different instances of drivers and CPU API libraries.

3.1.2 The URB Bridge

Each URB bridge is a gateway between URB CPU and nodes as illustrated in Figure 3.2. The connection between the URB CPU and bridges are called *uplink*, and connections between bridges and nodes are called *downlink*. The main purpose of a URB bridge is to buffer incoming asynchronous data and requests from the uplink and enabling data exchange with hardware nodes as the master of each I^2C downlink connection.

In order to achieve hardware modularity, URB bridge firmware is designed to incorporate three functional layers.

3.1.2.1 The Uplink Interface

The uplink interface subsystem implements a connection between the URB CPU and bridges, and detects asynchronous data and command transmissions over the uplink. Different uplink connection media (such as ISA, USB, RS232, etc.) require different instances of the uplink interface to handle uplink communication services. Incoming data from the uplink is received and buffered in this layer to be served according to priority and reception time.

3.1.2.2 The Job Controller

The job controller subsystem is responsible from classifying incoming uplink messages and providing appropriate responses to these messages. Each incoming

message is classified as either a bridge command, a node command or a node read/write operation. The job controller interfaces the downlink to the uplink, and schedules new events.

3.1.2.3 The Downlink Interface

The downlink interface subsystem provides a connection between the URB bridge and linked nodes. It also implements the I^2C based communication functionalities to master the downlink. Node requests and commands are directed to the downlink interface subsystem by the job controller and any responses from nodes are directed back to the job controller. Although the downlink interface is currently constrained to I^2C, other alternative implementations, such as the CAN bus are also possible.

3.1.3 URB Nodes

URB nodes are embedded devices interfacing with sensors and actuators. Each node is linked to a bridge and could be accessed over an I^2C bus. URB node firmware also offers a two layered structure: the downlink interface layer and the node application layer. The downlink interface layer implements I^2C communication details and downlink protocol details. In contrast, the node application layer manages domain specific sensory acquisitions and actuation actions.

A simple node library is provided to ease development of a node. This library implements protocol details and users are free to implement custom nodes according to their needs. A simple and standard API is also provided to users in order to offer a simple and fast development environment. The following section includes details of node implementation.

3.2 Functional Requirements

The URB is a modular, real-time, and reliable communication infrastructure designed for small robots as it was explained in previous chapters. In order to satisfy these needs, the URB communication protocol should implement some requirements and these requirements are listed as follows:

3.2.1 System Requirements

A modular communication protocol needs to be extended by adding new devices, namely nodes. New nodes need to be detected by the system in order to be included in the whole operation. Therefore, the URB protocol supports **auto-discovery** which is a method for discovering the network configuration and identifying devices that reside in a network[4]. The URB doesn't need to support hot-plugging, so discovery can be handled during the initialization of the bus.

Extending the network is limited in almost all communication protocols because of either electrical or protocol related constraints. For instance, in the USB specification[1], USB is limited up to 5 tiers and 127 devices for each tier. Analogously, the URB protocol implements exactly 2 tiers and 17 nodes can be connected to each bridge. However, no substantial limits should be imposed for the URB protocol. Extensions to the URB bus must be primarily limited by the available downlink and uplink bandwidths.

Besides modularity and extendibility, the URB protocol is also subject to strict real-time constraints. Hence, **latency** is a key constraint for its design. URB guarantees that a command sent by the main CPU is responded to in at most 2 ms. In other words, the latency between the time the CPU requests data from a node and the time it receives the information should be at most 2 ms.

One of the most important characteristics of the URB protocol is the **synchronization** of nodes. Each URB node implements its own clock and their

clocks drift away from each other even they are initiated at the same time. Therefore, periodic local data updates for each node is done at different time instants. URB protocol should also support a synchronization approach to solve this problem.

3.2.2 Hardware Requirements

Depending on the hardware configuration and types of μC used for devices, limitations of the whole communication system may change. However, for real-time purposes, the URB hardware should support 1Mbps raw communication speed for each downlink and 10Mbps raw communication speed for each uplink. These requirements might be relaxed for debugging purposes and soft real-time implementations.

Safety is another concept that needs to be taken into account for a dynamic autonomous robot. Each URB bridge supports shutting all actuator and sensor nodes in case of any emergency. URB bridges should have the ability to turn on and off all nodes associated with that bridge, so URB nodes receive power from their downlink connection.

3.2.3 Software Requirements

One of the primary purposes of the URB infrastructure is to provide a rapid development of a communication environment. In order to provide a simple development environment, there should be various libraries and APIs provided by the URB protocol. Therefore, URB should provide node libraries associated with a wide range of μCs and versatile, simple APIs should be also provided for fast and efficient development of nodes.

On the CPU side, a driver layer is needed for different uplink implementations such as RS232 and USB. These drivers should be supported for various operating systems such as Linux, Windows and QNX. Purpose of these drivers is to abstract

away details of low level uplink implementation. URB CPU APIs should be provided for fast CPU implementation, as well. However, implementation details of CPU libraries and APIs are excluded from the content of this thesis.

3.3 Autonomous Data Acquisition

Most of the data acquisition operations need to be executed periodically within robotic applications. For a mobile robot, sensory data and motor positions are good examples for periodic data acquisition and they are mandatory for safe, reliable motion. In case of a physically distributed architecture, this kind of data should be received from relevant components periodically over a communication medium.

Since URB is also developed for mobile robots, periodic data acquisitions constitute the largest proportion of data transmitted over the URB. When the URB CPU needs to retrieve data from sensory or actuator nodes, the CPU requests the data and receives the response to the request. In other words, each data acquisition is expected to be handled through a two way data flow over the URB. This situation brings a heavy workload over the URB and occupies the largest bandwidth. In order to reduce workload and bandwidth occupation, the URB protocol proposes a solution by automating periodic data acquisition operations.

For this purpose, the URB CPU may instruct a bridge for periodic data acquisition from a specified node and the bridge responds periodically with the requested data. In other words, the CPU does not put extra effort for requesting data periodically buy it only waits for the data to be read from pre-defined nodes. This way, the bridge does not need to control periodic data acquisition operations and the workload on the uplink and the CPU is significantly reduced. This means extra time for computational operations on the CPU and more bandwidth for other uplink transactions. Autonomous data acquisition is one of the most important contributions of this thesis together with node synchronization.

3.4 Node Synchronization

Physically distributed architectures bring various advantages to robotic applications. However, the primary problem in these architectures is the asynchrony in data acquisition operations. To address these problems, the URB protocol proposes a node synchronization approach.

In the absence of an explicit synchronization scheme, two important problems arise. First and foremost is the drift in periodic tasks over nodes. Each node has its own micro-controller and there are inevitable differences in clocks of different μCs. The differences in local clocks effect the periodicity of local data acquisition operations.

Second problem is caused by the timing of read operations. Assume that a node periodically performs data acquisition operations and the bridge reads the data just before the data is updated. This situation would cause a bridge to read a relatively old data, increasing data latency.

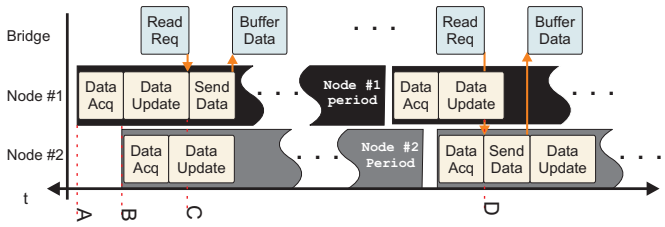

Figure 3.3: Timeline of events for a bridge and two *unsynchronized* URB nodes.

These two problems are illustrated in Figure 3.3 where bridge requests read operations from two different nodes with asynchronous periodic data acquisition. Node #1 issues a data acquisition at point A and Node #2 issues a similar request at point B. Even if each node were to be initiated at the same time, it is possible that local clocks would drift in time, illustrated by difference between points A and B. Points C and D also illustrate read events but at point D, the bridge sends a premature read request that cause the bridge to read stale data. This situation results in an increase in data latency.

In order to solve these problems, there are a wide variety of approaches published in the literature. Some of the synchronization algorithms and implementations are surveyed in Section 2.7. Among these approaches, the URB protocol implements an elegant node synchronization mechanism inspired from Phase Locked Loops.

3.4.1 Node Synchronization Algorithm

PLL is an old but efficient way of matching phases of two distinct signals. The primary idea behind this approach is to detect the phase and frequency of an input signal and try to minimize the difference between input and output signals by using a voltage controlled oscillator. Figure 2.6 illustrates the basic form of PLL. The URB protocol implements a PLL like approach in order to make nodes lock to a central, broadcast signal.

The first component of our approach is the broadcasting of a *downlink heartbeat* signal. The downlink heartbeat signal is broadcast by the bridge in the form of a simple node command. This command is transmitted periodically by a bridge and received simultaneously by all nodes linked to a downlink connection. This node command is used as an input signal providing a consistent timebase to each node and the idea is to synchronize each internal node heartbeat according to this periodic message. Figure 3.4 illustrates the synchronization scheme implemented by each node.

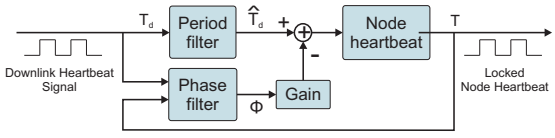

Figure 3.4: Block diagram for the downlink synchronization algorithm.

In this scheme, two simple filters are used to measure the phase of the downlink heartbeat and adjust the period of node heartbeat. The initial step of this scheme is to detect the period of the downlink heartbeat and measure the difference between the phases of the downlink and local heartbeats. Then, the node

heartbeat is adjusted appropriately based on the period of the downlink heartbeat signal and the phase difference between the downlink and node heartbeats. As a result, the node heartbeat converges to the downlink heartbeat.

In order to eliminate small variations in the period of the incoming downlink heartbeat and the effects of external noise, we used a simple filter to estimate the period of the downlink heartbeat

$$\hat{T}_d[k+1] = \hat{T}_d[k] - K_d(\hat{T}_d[k] - T_d), \tag{3.1}$$

where T_d is the most recently measured downlink heartbeat period and $\hat{T}_d[k]$ is the estimated heartbeat period. According to this filter, each node tries to estimate the reception time of the next heartbeat signal $\hat{T}_d[k+1]$. K_d is the predefined gain value used to reduce the effect of alternations in the downlink heartbeat. If this value is too small, changes in the downlink heartbeat would effect the estimated heartbeat period slowly and if its value is closer to 1 then these changes would effect the estimated period immediately.

After estimating the period of next heartbeat signal, the node heartbeat is adjusted based on the difference between heartbeat signal and node heartbeat according to the equation

$$T[k+1] = \hat{T}_d[k+1] - K_n((t_{nhb} + \Delta t_{acq}) - t_{dhb}), \tag{3.2}$$

where T[k+1] is the period value to be used for the next node heartbeat cycle. t_{dhb} is the time that downlink heartbeat signal is received and t_{nhb} stands for the time that node heartbeat cycle starts. Each node should also be aware of the duration of its own data acquisition in order to adjust its heartbeat, so that data acquisition events end before the downlink heartbeat signal received. As a result, Δt_{acq} indicates an upper bound on the duration of data acquisition for a node and is included in this calculation. Δt_{acq} varies in each cycle and nodes keep track of its largest value in order to guarantee that data acquisition events end well before downlink heartbeat reception. K_n is also a predefined gain value to reduce the effect of variations in Δt_{acq}.

As a result of this algorithm, the downlink heartbeat signal from a bridge prior to read requests guarantees the bridge to retrieve the results of most recent data

acquisition events. The next downlink heartbeat causes nodes to trigger data acquisition events and update older data. Therefore, node data between two downlink heartbeat signals is said to be synchronized and the bridge should read before the next downlink heartbeat signal to preserve data synchrony. Figure 3.5 illustrates the results of this algorithm and improvements to the situation illustrated in Figure 3.3.

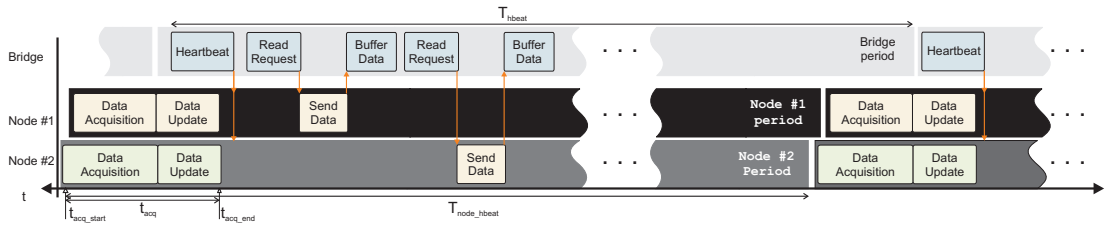

Figure 3.5: Timeline of events for a bridge and two *synchronized* URB nodes.

3.5 URB Nodes and The Downlink Communication Protocol

3.5.1 Downlink Communication Model

In URB, nodes are linked to the associated bridges over an I^2C bus and each node is configured as a slave to the associated bridge (master). Being masters of a downlink connection, each transaction is initiated by bridges and related nodes respond to these requests.

URB nodes implement a *message box* structure similar to USB device endpoints [1]. Each node is allowed to have 8 *outboxes* and 8 *inboxes*, each of which are internally double buffered as illustrated in Figure 3.6. *Outbox* is the message box type used to send data out. In contrast, nodes use *inboxes* to receive input in the form of a command or data from bridges. In other words, *inbox* is the place where a bridge writes a message to and *outbox* is the place where a bridge reads related data from. Each message box is initialized by the user with a fixed static size and used by the URB libraries for communication transactions.

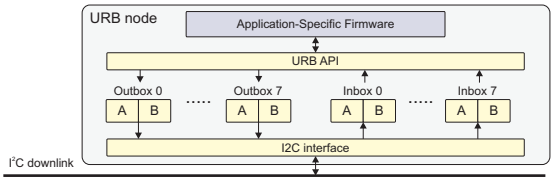

Figure 3.6: Protocol details of a URB node. Each node supports a total of 16 (8 outboxes and 8 inboxes) double buffered message boxes.

Each downlink transaction includes an address byte in order to identify interested nodes. This address information includes 4 bits node address and 3 bits message box address. Nodes receiving the the first byte of incoming transaction use this information in order to accept incoming data and relate the data to one of the message boxes according to message box address. By using 4 bits for address definition enables us to define 16 different nodes for each bridge. However transactions with node address 0 is accepted by all nodes. In other words, node address 0 is reserved for broadcasted transactions. 3 bits of message box data define one of the 8 outboxes or inboxes and read/write transactions are identified by a single bit.

3.5.1.1 Built-in Message Boxes

Each URB node implements 8 inboxes and 8 outboxes as described above. Among these message boxes, 2 are unique for each node and used for special purposes. Outbox 0 is used to store node configuration information and Inbox 0 is used for receiving node commands from bridges.

Outbox 0 is defined by the node API during node initialization and is used to provide information about node configuration and attributes. Bridges should read the information stored in this outbox to discover node configuration information. Table 3.1 illustrates the layout of Outbox 0.

In URB, each node is expected to provide a specific functionality, such as distance sensing or motor control. The **node class** identifier defines the functionality of a node. Nodes with the same functionality are further identified by **node indexes**. The **state** field is used to display the current state of a node such

outbox 0

7	6	5	4	3	2	1	0
node class							
state				node index			
-	C			ver		rev	

Table 3.1: Outbox[0] fields and layout.

as *uninitialized*, *idle*, *active* or *reserved*. Version and revision information about the firmware deployed on a node is defined in **ver** and **rev** fields. Finally, the **C** bit is used as a flag in order to define if the node supports compressed read and write operations. Compressed transactions are designed to improve the overall transmission performance and will be described in Section 3.5.2 in details.

The second built-in message box of nodes is the Inbox 0. This inbox is used to receive protocol commands from bridges or the URB CPU. Table 3.2 illustrates the layout and fields of Inbox 0 structure:

inbox 0

7	6	5	4	3	2	1	0
opcode					reserved		
argument[0]							
argument[1]							

Table 3.2: Inbox[0] fields and layout.

The URB protocol uses inbox 0 to send predefined protocol commands to nodes by using built-in *operation codes* (opcode). Each opcode and related arguments defines a specific task for nodes and is written to related fields of Inbox 0. Reception of a protocol command invokes associated procedures over a node. Table 3.3 defines the node commands and details of each command.

- **NOP**: This command is a dummy command that has no effect.
- **SIGNAL**: This command is used to send an asynchronous signal to nodes. Specifics of this command should be implemented by the node application layer. Both URB CPU and nodes should agree on the semantics of signals. One byte argument is used to parametrize the behaviour of this command.

Opcode	Command	Arg[0]	Arg[1]	Description
00000b	NOP	N/A	N/A	No operation
00001b	SIGNAL	BYTE	N/A	Asynchronous signal
00010b	SYNC	N/A	N/A	Synchronization signal
00011b	SYNC_LOCK	BYTE	N/A	Enable/disable locking to SYNC
00100b	ACTIVATE	N/A	N/A	Activate data update
00101b	DEACTIVATE	N/A	N/A	Deactivate data update
11111b	RESET	N/A	N/A	Reset node

Table 3.3: Opcodes and arguments for node commands.

- **SYNC**: The concept of synchronization was explained in Section 3.4. Briefly, this command is used as a central heartbeat to enable synchronous data acquisition across nodes. Any bridge that needs to synchronize its nodes should broadcast this signal periodically, so that nodes can synchronously update data acquisition.

- **SYNC_LOCK**: Initially each node starts local update periods depending on their own clock cycles and each node has an independent update period from other nodes. This command is used to enable or disable locking to the heartbeat signal broadcasted by any bridge. Details of this command are examined in Section 3.4.

- **DEACTIVATE**: By default, each node boots up with periodic local updates enabled. This command is used to *deactivate* periodic update operations within a node.

- **ACTIVATE**: This command is the opposite of DEACTIVATE command and activates periodic updates within a node.

- **RESET**: This command resets the states and all used variables of the firmware and brings the node to a state identical to the state immediately after boot-up.

3.5.1.2 Double Buffering Approach

Generally, embedded systems require small response times with no tolerance for busy-waiting. Therefore embedded systems usually implement an *interrupt* mechanism to detect and respond to external events [43]. An interrupt is an asynchronous signal from hardware, indicating the need for attention. For instance, a serial port chip needs attention when it receives a character and raises an interrupt indicating the need for attention. When the microprocessor detects an interrupt, it stops what it was executing and switches context to start executing an *interrupt service routine* (ISR).

Interrupts and ISRs are essential for multitasking, especially for real-time computing. However, interrupts need to communicate with the rest of the code as well. ISR and task code, most of the time, share variables which may cause *shared-data problems*. Suppose that in a program, task code and an ISR share a variable which is wider than one byte. Task code or ISR may want to manipulate the variable while the other one reads the variable. If task code is in the middle of a process related with the variable, an incoming interrupt would stop execution of task code and invoke related ISR. In this scenario, switching context to the ISR would result erroneous variable values either for the ISR or for the task code.

One of the methods used for solving a data sharing problem is to disable interrupts until task code completes its task with the variable. Yet, this solution results in a potentially long delay in responding to interrupts. Hence we use a double buffering approach for both inboxes and outboxes. Double buffering is a widely used technique for minimizing the delay in input/output operations.

In order to safely use double buffering, each inbox and outbox contains seperate read and a write buffers with identical size. If a message box is in use, it is **locked** by that task so that variables stored in the buffer are guaranteed to stay the same. When a task finishs its job, it **releases** the message box. Data in a message box is updated by writing over the write location of the double buffer and write/read buffers are switched if the message box is not in use. Figure 3.7 and 3.8 illustrate our double buffering state machine for inboxes and outboxes

relatively.

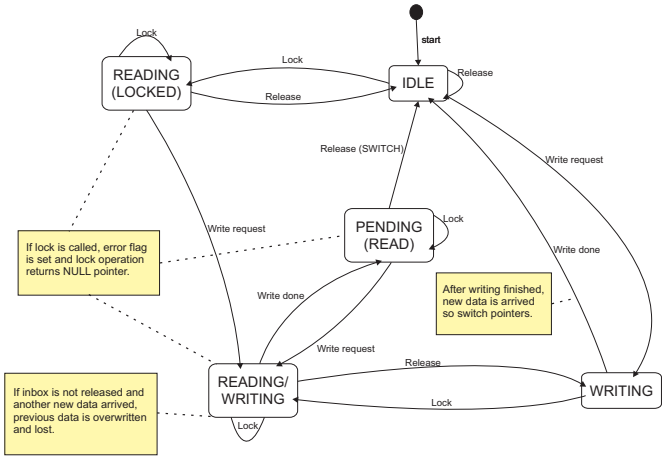

Figure 3.7: State diagram for double buffered inboxes.

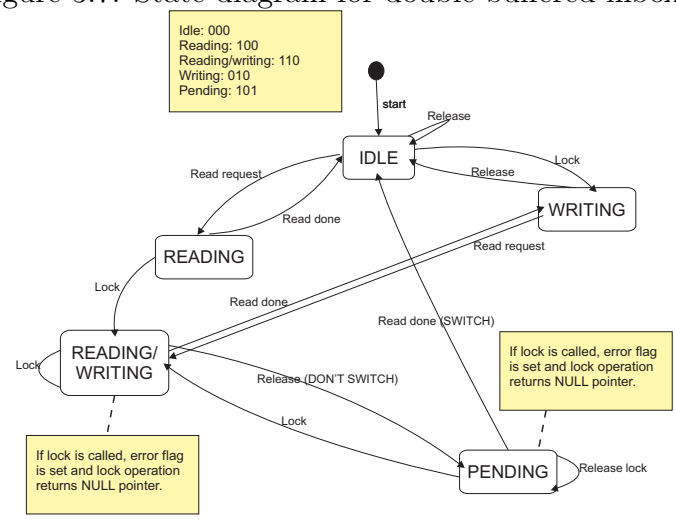

Figure 3.8: State diagram for double buffered outboxes.

3.5.1.3 Downlink Transactions

Conventional microcontrollers that support I^2C/SMBUS transactions have some differences in the way they implement transactions. Some of these microcontrollers allow software decoding of addresses. On the other hand, some microcontrollers decode address information in hardware and checks with a preselected node address. For the purpose of supporting both these implementations, we have defined 6 different transaction types. Packet formats and details of these transaction types are listed below:

- **Default Reads**: Default read transactions are used to read from the default outbox. Default outbox is defined by each node internally and it is set to Outbox 0 after bootup or reset. The default outbox is changed to the last read outbox after a targeted read.

  This transaction type results in substantial bandwidth savings, especially for nodes that don't support software address decoding, especially if only a single outbox is continuously and periodically read. Table 3.4 illustrates the sequence of data exchange during a default read operation.

Byte Count	7	6	5	4	3	2	1	0	Direction
1	node address				0 0 0			1(R)	B → N
2	content byte[1]								N → B
...									
N+1	content byte[N]								N → B

Table 3.4: I^2C data transactions for default read.

- **Targeted Reads**: This type of read transaction is defined and used especially for nodes that do not support software address decoding. This transaction is simply a write request followed by a read request. Firstly a write transaction is issued to set the new outbox that is to be read. Secondly a default read transaction is issued to read from previously set outbox. Table 3.5 describes data exchanges for a targeted read transaction. As illustrated in Table 3.5, the second byte of the transaction defines the outbox and very first bit of this byte defines the subsequent operation (i.e. if it is a targeted read or targeted write transaction).

Byte Count	7	6	5	4	3	2	1	0	Direction
1	node address				0 0 0			0(W)	B → N
2	1	reserved				message box #			B → N
3	node address				0 0 0			1(R)	B → N
4	content byte[1]								N → B
...									
N+3	content byte[N]								N → B

Table 3.5: I^2C data transactions for targeted read.

- **Compressed Reads**: As described above, some microcontrollers support

software address decoding. This allows us to embed the message box number into a single byte within the node address information. This implementation eliminates the inefficiency of read transactions and saves us significant bandwidth. Table 3.6 illustrates compressed read requests.

Byte Count	7	6	5	4	3	2	1	0	Direction
1	node address				message box #			1(R)	B → N
2	content byte[1]								N → B
...									
N+1	content byte[N]								N → B

Table 3.6: I^2C data transactions for compressed read.

- **Targeted Writes**: Targeted write transactions are similar to targeted read transactions but we do not need to change transfer mode after writing the message box number to be used. So, a message box number is transmitted after node address and data is directly written to the specified message box. Table 3.7 describes details of targeted write transactions.

Byte Count	7	6	5	4	3	2	1	0	Direction
1	node address				0 0 0			0(W)	B → N
2	0	reserved				message box #			B → N
4	content byte[1]								B → N
...									
N+2	content byte[N]								B → N

Table 3.7: I^2C data transactions for targeted write.

- **Compressed Writes**: Microcontrollers supporting software address decoding also allow us to increase performance of write transactions and save bandwidth. Similar to compressed read transactions, the message box number is encoded in the first byte of downlink messages within node address information. Table 3.6 illustrates compressed write transactions.

- **Broadcast Writes**: The main difference between broadcast write transactions and the write transactions described above is their node addresses. Write transactions with node address 0 are accepted by all nodes, namely they are *broadcast requests*.

Byte Count	7	6	5	4	3	2	1	0	Direction
1	node address				message box #			0(W)	B → N
2	content byte[1]								B → N
...									
N+1	content byte[N]								B → N

Table 3.8: I^2C data transactions for compressed write.

Broadcast write transactions are used to send a command to all nodes at the same time. For example, the downlink heartbeat message is one and most important broadcast transaction within the URB protocol. It is used to synchronize all nodes linked to a downlink connection (Section 3.4).

3.5.2 Downlink API Details

The URB communication protocol provides a simple and versatile API in order to facilitate rapid deployment of URB nodes. Users should use related libraries and API in order to develop application specific firmware on URB nodes. Currently, URB node API supports 8051 based microcontrollers from Silicon Libraries.

The URB node API is flexible enough for all types of firmware development and associated libraries provide all necessary components for node development. Some of the functions are implemented by the API and are ready to be used by node developers. However, it is important to provide a flexible node implementation, since node firmware implementations would differ according to user and implementation needs. So, some of the core methods are left for the user to implement. These two types of functions are listed under below sections. A sample node firmware implementation is also provided in Appendix B.

3.5.2.1 Functions Implemented by the URB Node API

This section describes usage and functionalities of methods provided by the URB node API. These methods are implemented by the node libraries such that all

implementation details are abstracted from the user. They handle local configuration of nodes and all communication details. These functions are listed as follows:

- `void URB_SetAddress( byte i2c_address )`

  This function is used to set node address to be used as the protocol address of the node. This address is defined with 4 bits and is supplied with the least significant 4 bits of input argument *i2c_address*.

- `void URB_SetClass( byte class )`

  This function sets the class of the URB node. Node class specifies the functionality of nodes such as sensor and actuator nodes.

- `void URB_SetIndex( byte index )`

  This function sets the node index, used to identify nodes with the same functionality (i.e. same class).

- `void URB_SetVersion( byte version )`

  This function is used to set the version of node firmware.

- `void URB_SetRevision( byte revision )`

  This function is used to set the revision of node firmware.

- `bool URB_SetupInbox( byte boxID,`
  `byte size, byte xdata *head, bool overwrite )`

  The URB node API gives great flexibility with user-defined message box sizes and message box behaviors. This function is used by API users to setup inboxes except Inbox 0 since it is internally configured and reserved to be used by the URB protocol. The user should provide a pointer to an external memory area that is twice th size of needed area. This is needed to implement double buffering and used internally. Even the area should be twice the size of inbox size, input argument *size* should indicate the size of the inbox. The *overwrite* flag is used to determine whether a buffer overflow results when new data overwrites old data on the incoming message box.

- `bool URB_SetupOutbox( byte boxID, byte size, byte xdata *head )`

  Similar to *URB_SetupInbox*, this function is used to configure node outboxes except Outbox 0 since it is configured internally by the URB node API.

- `byte* URB_LockInbox( byte boxID )`

  URB message boxes implement a double buffered approach in order to prevent data loss and shared data problems as described in Section 3.5.1.2. This function locks the inbox specified with *boxID*, preventing double buffering code from swapping and returns a pointer to the specified inbox that is to be read by the user. This function **must** be used to access an inbox for reliable data operations.

- `byte* URB_LockOutbox( byte boxID )`

  Similar to URB_LockInbox function, this function is used to lock outboxes. It **must** be called before writing to an outbox.

- `void URB_ReleaseInbox( byte boxID )`

  This function releases (un-locks) a locked inbox and it **must** be called once a read operation is completed.

- `void URB_ReleaseInbox( byte boxID )`

  This function releases (un-locks) a locked outbox and it **must** be called after a write operation is completed.

- `void URB_UpdateDone( void )`

  URB nodes periodically update internal data specified by the user. Update sequences are application dependent activities and are defined by user as described in Section 3.5.2.2. For synchronization purposes, nodes need to know the time cost of update operations. So this function is used by the URB libraries to calculate the time duration of an update activity. This function **must** be called subsequent to the completion of update events, but the user should be aware of the place it is used. See app_update() definition in Section 3.5.2.2.

3.5.2.2 Firmware Dependent Functionalities

The URB node API facilitates node configuration and communication but it also gives the flexibility of defining firmware dependent activities. Functions described below are used to define firmware related functionalities and should be implemented by users. These functions are internally called by node libraries and define the mechanism of each node.

- `void app_bootup( void )`

  This function is called when the microcontroller boots boots up. It is responsible for configuring system clock, pins, submodules (such as ADC, serial communication submodules), interrupt sources and setting up node address, class, index, version, revision, and message boxes. Users could configure nodes depending on their needs and operation specific events.

- `void app_fault( byte errno )`

  The URB node allows users to catch and handle communication errors. This function is used to define the behavior of nodes in case of any errors.

- `void app_idle( void )`

  This function is called continually by the URB node libraries within the main loop when there is nothing better to do. It does not guarantee periodicity but is called as frequently as possible. The example node implementation in Appendix B illustrates the usage for this function. It is better to use this function for checking events that do not require periodicity such as checking inboxes for incoming data.

- `void app_init( void )`

  This function is called immediately after the first boot up or soft reset command issued by bridge. It allows nodes to prepare all internal data structures needed by the application and to enable interrupt sources. After a soft reset, this function clears all the application data and brings the application state to its initial value after first boot-up.

- `void app_reset( void )`

  This function is called after receiving a reset command from the bridge and it should restore submodules used by the application to their initial states. Subsequent to this function, *app_init* is called to reinitialize relevant components of the node application.

- `void app_signal( byte signo )`

  This function is used as an instant messaging service for user-defined tasks. As described in Table 3.3, signal message is directly written to Inbox 0 as a node command and one byte argument parametrizes the reaction of the node to this message. After receiving a signal command, this function is called by node libraries and should handle this message as defined by the user.

- `void app_update( void )`

  This function is called with a constant frequency by node libraries in order to update acquired data values. An example usage of this function would be periodic data acquisition as illustrated in Appendix B.

  app_update function plays a key role in synchronization of nodes. The calling frequency of this function changes according to the downlink heartbeat to match with the phase and frequency of heartbeat message. Subsequently, *URB_UpdateDone* is called in order to measure time cost of update operations and the time cost is used for synchronization of nodes. See Section 3.4 for detailed explanation of synchronization.

3.6 URB Bridges and The Uplink Communication Protocol

3.6.1 Uplink Communication Model

As described in Section 3.1.2, the URB uplink connections link the URB CPU to bridges. Uplink communication establishes a bi-directional stream of bytes and can be categorized based on the direction of data transactions. **Job Dispatch Packets** (JDPs) are incoming messages to bridges sent by the CPU. Similarly, **Job Response Packets** (JRP) are sent by bridges to the CPU in response to JDPs. Both job dispatch and job response packets can hold a maximum of 32 bytes of data content. Packet formats and details for each packet type are discussed in this section.

Uplink communications can be implemented through a variety of channels, such as RS232, USB, or ISA bus. Currently, the URB uplink communication is implemented for both RS232 and USB. However, in this thesis we only investigate RS232 based uplink communication.

3.6.1.1 Job Dispatch Packets (JDP)

JDPs are transmitted by the URB CPU to dispatch new tasks for either the bridge or nodes. Bridges receiving a JDP, identify the type of the request by considering the most significant bit of the packets' first byte. This byte also defines two additional flags: **UR** and **RR** flags. The UR flag is used to define whether the associated task is urgent and RR flag defines whether a response for the associated task is required. Beside these flags, the first byte of a JDP includes the size of the packet content. Table 3.9, Table 3.10 and Table 3.11 illustrate the detailed definitions of these packets.

Node requests initiate a data transfer to or from a node by sending a node command to the related bridge. The second byte of node request packets are

7	6	5	4	3	2	1	0
0	RR	UR	packetsize - 2				
Node address				msgbox #			0
Packet Content (32 bytes max.)							

Table 3.9: Node request packet format for write operations.

7	6	5	4	3	2	1	0
0	RR	UR	packetsize - 2 (**1**)				
Node address				msgbox #			0
Outbox size							

Table 3.10: Node request packet format for read operations.

similar except the last bits. The last bit for each of these two packet types is used to identify if this operation is a read or write. The second byte also includes information for the target of data transmission: the node address and the message box number. A 4 bit node address allows us to support up to 16 nodes for each bridge and a 3 bit message box number supports up to 8 outboxes and 8 inboxes. If the node request is a write operation, the packet contents include data to be written to a specified node and inbox. The URB protocol supports write operations to transmit up to 32 bytes of data in a single packet. But if the operation is a read request, packet content is fixed to one byte, defining the size of the message box that is to be read.

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2				
Bridge Command							
Command Args. (32 bytes max.)							

Table 3.11: Bridge request packet format.

Bridge requests, in contrast, encode data and command exchanges between the CPU and the bridge. The second byte of the packet indicates the bridge command type defined by the URB protocol. Appendix A provides a list of bridge request types, brief description for these requests, command arguments and responses for each command. Section 3.6.2 also provides a detailed description for all requests.

3.6.1.2 Job Response Packets (JRP)

Upon reception of any job dispatch packet, the bridge is expected to send a response packet to the URB CPU if a response is expected. Some of the JDPs are set to send a response packet by default (see Section 3.6.2). However, the CPU indicates that a response to a dispatch packet is needed by setting the RR field in the JDP.

Node response packets contain the results of a node read or write operations. However, bridges can initiate a read operation from nodes autonomously and send response packets to the CPU as described in Section 3.3. In order to distinguish node response packets for explicit node requests from autonomous responses generated by bridges, JRPs contain a field called Autonomous Response (**AR**). If the response packet is the result of an explicit response, then the AR bit is cleared (0), otherwise it is set. Table 3.12 and Table 3.13 illustrate both read response and write response packets.

7	6	5	4	3	2	1	0
0	AR	UR	packetsize - 2				
URB node addr.				msgbox #			1
Content Byte 1							
...							
Content Byte N							

Table 3.12: Node response packet format for read operations.

Response packets to node read operations contain the most recently acquired data in the specified outbox. Both the CPU and the addressed node have common knowledge of the content of the data located in a read response packet. The data size for a read response packet can be at most 32 bytes.

7	6	5	4	3	2	1	0
0	AR	UR	packetsize - 2				
URB node addr.				msgbox #			0
X	X	X	X	X	X	X	ACK

Table 3.13: Node response packet format for write operations.

Response to a node write operation contains only one byte data indicating the

status of the write operation as illustrated in Table 3.13. The least significant bit (LSB) of the response packet (ACK) is set to indicate that the write operation is successful, it is cleared otherwise.

Responses to bridge request packets contain the corresponding response to the related bridge command as illustrated in Table 3.14. Bridge response packets do not contain any record of the bridge request packet. Since bridge request packets are handled sequentially by bridges, the CPU should keep track of the order in which bridge request are sent.

7	6	5	4	3	2	1	0
1	ER	UR	packetsize - 2				
Response Content (33 bytes max.)							

Table 3.14: Bridge response packet format.

Note that bridge response packet format does not include the AR field in the first byte. The AR field is used to identify autonomous node response packets, so bridge responses do not need such a field. Instead, bridge response packets include the **ER** field in order to report erroneous situations. If the ER bit is set (1), the packet content provides a single byte indicating the error type.

3.6.2 Bridge Commands and Uplink Protocol Details

This section presents all the details of bridge commands and their application.

- **BRG_RESET_CMD**

  This command resets the bridge so that it can recover from any erroneous conditions. It requires no input arguments and returns no response even if RR bit is set. Packet structure for this command is illustrated in Table 3.15.

- **BRG_GETVER_CMD**

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (0)				
BRG_RESET_CMD (0X00)							

Table 3.15: Packet structure for BRG_RESET_CMD.

This bridge command is used for receiving the version and revision of the firmware running on the bridge. Response packet includes the version and revision data in two separate bytes. The RR flag should be set to receive a response to this command. Packet structure for this command is illustrated in Table 3.16.

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (0)				
BRG_GETVER_CMD (0X01)							

Table 3.16: Packet structure for BRG_GETVER_CMD.

- **BRG_CLOCK_CMD**

  Each bridge includes its own timer and generates a tick at 10 KHz. This command is used for reading this tick value in 4 bytes. The RR flag should be set if a response is requested. Packet structure for this command is illustrated in Table 3.17.

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (0)				
BRG_CLOCK_CMD (0X02)							

Table 3.17: Packet structure for BRG_CLOCK_CMD.

- **BRG_LED_CMD**

  This command can be used to turn on/off LEDs located on the bridge. It requires only one argument indicating the next state of the specified LED and returns the previous state of the LED. The most significant 7 bits of the argument is used for defining LED number and the least significant bit defines the next state of the LED. Packet structure for this command is also illustrated in Table 3.18.

- **BRG_PKTCOUNT_CMD**

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (1)				
BRG_LED_CMD (0X03)							
LED Number							(0/1)

Table 3.18: Packet structure for BRG_LED_CMD.

Bridges count packets received from the CPU and packets sent to nodes over the downlink connection. This command returns the packet count data in 8 bytes. First 4 bytes are packets received from the uplink and last 4 bytes are packets sent over the downlink. Packet structure for this command is illustrated in Table 3.19.

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (0)				
BRG_PKTCOUNT_CMD (0X04)							

Table 3.19: Packet structure for BRG_PKTCOUNT_CMD.

- **BRG_DISCOVER_CMD**

  When the bridge boots up, it is not aware of linked nodes and information for the nodes connected to the downlink. This command is used to discover nodes linked to the bridge. It is called with no arguments as illustrated in Table 3.20 and it returns two bytes as the result of the discovery. Bridges always respond to this command with two bytes even the CPU does not request a response (RR=0). The most significant bit of the first received byte indicates the result for node with address 16 and the least significant bit of the second byte indicates address 0. Interim bits are ordered from adresses 15 to 1. Node address is used for broadcast downlink transactions.

  Discovery operation causes the bridge to search through the whole node address space except 0 (1 through 16). The discovery operation is simply a node read operation from outbox 0. As a result, bridges acquire node information from active nodes and store them for further CPU requests of node information. As noted, if a node is removed or added to the URB downlink connection, a new discovery should be initiated to detect modifications.

- **BRG_NODEINFO_CMD**

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (0)				
BRG_DISCOVER_CMD (0X05)							

Table 3.20: Packet structure for BRG_DISCOVER_CMD.

This command is used to get node information for a specific address and structure of this command packet is illustrated in Table 3.21. After receiving this command, the bridge responds with the 4 byte node information packet (address, class, index, version and revision) received from nodes during the last discover operation. If information for a node address that is not found during last discovery is requested, then bridge responds with four bytes of 0 as the node information. So, if the BRG_DISCOVER_CMD was not received before this command, there will not be any recorded node information on the bridge and it will return 0 for all addresses.

NOTE: BRG_DISCOVER_CMD should be sent before this command in order to read correct information for nodes. The BRG_DISCOVER_CMD command should be re-sent in order to update node information (for instance, in the case of new node addition).

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (1)				
BRG_NODEINFO_CMD (0X06)							
Node address (0x01 to 0x0f)							

Table 3.21: Packet structure for BRG_NODEINFO_CMD.

- **BRG_AUTOMATE_CMD**

  URB supports synchronization of nodes and periodic data acquisition from nodes without intervention from the CPU. This functionality is fulls controlled by bridges and this command is used to enable autonomous synchronization and autonomous data acquisition from nodes. This command requires one byte argument defining the flags for autonomous synchronization and data fetch messages and the frequency of the downlink heartbeat messages as multiples of 10 KHz. The packet structure for this command is presented in Table 3.22. The AS field is used to enable/disable autonomous synchronization messages and the AF field enables/disables autonomous

data fetch operations. The remaining six bits of the command argument defines the period of the downlink heartbeat signal. The response to this command is an acknowledgment byte with value 1 for setting automation operation correctly and 0 otherwise.

NOTE: In order to activate autonomous data acquisition (auto_fetch), BRG_DISCOVER_CMD should be sent beforehand.

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (1)				
BRG_AUTOMATE_CMD (0X07)							
AS	AF	Freq. of auto-sync. msg.					

Table 3.22: Packet structure for BRG_AUTOMATE_CMD.

- **BRG_AUTOFETCH_CMD**

  This command is used to inform bridges about nodes and message boxes that are going to be read autonomously. It requires 3 arguments for defining node address, message box ID and size, period and offset of this autonomous data fetch operation as illustrated in Table 3.23. Responses to this command return the ID of the defined autonomous data fetch operation. Autofetch IDs start from 0 and go up to 15. The returning ID should be saved so that bridges could use it later to cancel a previously defined auto-fetch operation.

  Two important arguments for this command are period and offset. The former is used to define the period of autonomous data acquisition in terms of ticks (1 ms). The other one is the offset value used to shift data acquisition in terms of ticks. For instance, if period is set to 5 and offset is set to 1, related autonomous data acquisition is handled in every 5 ms starting after 1 ms.

  NOTE: This command should be issued at least once before starting autonomous data acquisition in order to define the node and the message box that is going to be read autonomously.

  NOTE: Maximum number of BRG_AUTOFETCH_CMD commands that can be accepted by a bridge is 16. More than 16 requests results in an error.

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (3)				
BRG_AUTOFETCH_CMD (0X08)							
Node addr.				msgbox #			0
Messagebox Size							
Period				Offset			

Table 3.23: Packet structure for BRG_AUTOFETCH_CMD.

- **BRG_CANCEL_AUTOFETCH_CMD**

  This command is used to remove previously defined autonomous data acquisition operation from the bridge queue. It requires the ID of the AUTOFETCH operation received after the associated BRG_AUTOFETCH_CMD as an arguement (Table 3.24). So the CPU should keep track of previously issued autonomous data fetch operations to be able to cancel them. The bridge also responds with 1 if cancellation operation is successful, 0 if otherwise.

  As an example, assume that the CPU requests 3 autonomous read operations and receives IDs 0, 1 and 2. If the CPU cancels operation with ID 2, and requests a new auto-fetch operation, the bridge would assign this request to ID 2 but indexes it after ID 3. Besides, if there is no available slot to assign a new auto-fetch operation, the bridge responds with a value 0xFF.

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (1)				
BRG_CANCEL_AUTOFETCH_CMD (0X09)							
Autofetch ID (0 to 15)							

Table 3.24: Packet structure for BRG_CANCEL_AUTOFETCH_CMD.

- **BRG_NOP_CMD**

  This command is a dummy command and is ignored by bridge (Table 3.25).

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (0)				
BRG_NOP_CMD (0X17)							

Table 3.25: Packet structure for BRG_NOP_CMD.

- **BRG_TOKEN_CMD**

  This command transmits a response packet with a special 4-byte token (0xff, 0xa5, 0xb9, 0x9f) back through the uplink. The purpose of this command is error detection and error recovery. (Table 3.26)

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (0)				
BRG_TOKEN_CMD (0X18)							

Table 3.26: Packet structure for BRG_TOKEN_CMD.

- **BRG_ECHO_CMD**

  This command forces the bridge to directly send the received data argument back over the uplink. Since response to this command doesn't fit into the generic response format defined in Section 3.6.1.2 and Table 3.14, it is very dangerous to use this command.

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (1)				
BRG_ECHO_CMD (0X19)							
Data to be echoed							

Table 3.27: Packet structure for BRG_ECHO_CMD.

- **BRG_DL_GETVER_CMD**

  This command is used to receive the version of the downlink subsystem and it includes version, revision and type string of the currently used downlink on the bridge (for instance *"i2c"*). It does not require any arguments and it responds with 10 bytes of downlink version information.

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (0)				
BRG_DL_GETVER_CMD (0X20)							

Table 3.28: Packet structure for BRG_DL_GETVER_CMD.

- **BRG_DL_CUSTOM_CMD**

  This command is used to send a custom command for the downlink subsystem on the bridge. The number of arguments and response packet size vary according to the implementation.

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (VAR)				
BRG_DL_CUSTOM_CMD (0X39)							

Table 3.29: Packet structure for BRG_DL_CUSTOM_CMD.

- **BRG_UL_GETVER_CMD**

  This command is used to receive version data for the uplink subsystem on the bridge and it includes version, revision and type string of currently used uplink (such as *"rs232"*). It does not require any arguments and it responds with 10 bytes of uplink version information.

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (0)				
BRG_UL_GETVER_CMD (0X40)							

Table 3.30: Packet structure for BRG_UL_GETVER_CMD.

- **BRG_UL_CUSTOM_CMD**

  This command is used to send a custom command for the uplink subsystem of the bridge. The number of arguments and response packet size vary according to the implementation.

7	6	5	4	3	2	1	0
1	RR	UR	packetsize - 2 (VAR)				
BRG_UL_CUSTOM_CMD (0X59)							

Table 3.31: Packet structure for BRG_UL_CUSTOM_CMD.

Chapter 4

Discussions on URB Performance

In this chapter, we describe some of the mostly used fieldbus applications, algorithms used for these applications and implementation details of the URB. This chapter concerns discussions on the URB protocol performance and implementation details compared to previously discussed applications.

4.1 Event Based vs. State Based

In event based fieldbuses, events are the source that triggers the initiation of data transmissions. Nodes detecting an event that causes a shift in the local node state, should employed to inform the central processing unit. According to this scenario, event based fieldbuses provide efficient use of bandwidth and node buffer space by transmitting only crucial data.

Hence, each message is important in terms of robustness and event based networks are not tolerant to any errors and loss in data transmission[38]. It is important to check whether a message is delivered successfully and re-transmit the same data in case of an error. Accordingly it is difficult to predict maximum number of messages transmitted from a node.

Besides, event based network protocols are inefficient under heavy network

loads, like alarm conditions for several nodes. In case of an alarm condition, each node tries to transmit at the same time instant and competes for communication channel. Therefore, it is difficult to predict the latency of a message delivery and contain adequate bandwidth for such a situation.

On the other hand, state based network protocols check each node in certain time intervals and can detect changes in internal state of nodes. Although this causes periodic messaging and bandwidth consumption, it is obvious that state based protocols are more predictable and tolerant to message loss than event based protocols. If a message is lost or erroneous, it is probable that this value is transmitted correctly in the next cycle with a predictable delay. State based systems can tolerate message loses by sending messages twice or third times faster than required in order to ensure the real time constraints. Also state based approach solves the race conditions during an alarm within the system by nature.

However, it is difficult to detect transient data over a state based protocol. Transient data is the term used for a data that is valid for a short time period and temporary. If a data within a node changed two times between two read cycles, main CPU would not be aware of the interim state of the node.

In the light of these discussions, it is simple to organize data transmission within a state based protocol and eliminating the effects of erroneous data by sacrificing the network bandwidth. Therefore, the URB protocol implements a state based approach in order to increase performance and reliability by avoiding the race conditions during hazardous circumstances.

4.2 Efficiency

Communication traffic for embedded systems is usually composed of short and periodic messages. Therefore, it is very important to use network bandwidth efficiently. Network *efficiency* can be defined as data bits delivered compared to raw network bandwidth[48]. Two factors characterizing network efficiency are packet overhead and media access overhead.

4.2.1 Packet Overhead

Packet overhead is all non-data bits added by the protocol to ensure proper communication and reliable transportation. These non-data bits might be CRC bits, address bits and acknowledgment bits. Network efficiency can be improved by reducing packet overhead. Hence, the URB protocol is designed to eliminate most of the extra bits as long as it is possible to do so.

There is a tradeoff between reliability and efficiency and decision is left for the user in most of the cases. As it is described in Section 4.1 user has the freedom to define a CRC algorithm and include this into the protocol for reliability. Other efficiency metrics are defined by the protocol and tried to be improved wherever it is possible.

Packet overhead for the URB protocol can be investigated as uplink overhead and downlink overhead, since uplink and downlink communication implement distinct protocols. Figure 4.1 illustrates package overhead for uplink communication. It can be easily observed that uplink efficiency is increased with the packet content size for both bridge commands and node requests. It is similar for responses of these packets. It is the user's responsibility to increase uplink efficiency by defining message box sizes in an optimal way. Since node requests and responses to these request compose the main part of the protocol, users should be aware. However, bridge requests are defined by uplink protocol and they do not depend users or application.

The URB uplink protocol defines 15 bridge commands and most of them do not have any arguments. If it is assumed that each bridge command is sent once in a second, 37 bytes should be transmitted over uplink media. 59.46% of these packet bits are actual data bits and this results reduction in the uplink ecciency. However, bridge commands are not periodic and frequent messages so they reduce uplink efficiency in excessive amounts. With a similar approach, responses to bridge commands include 79.03% of actual data bits which would be considered as a adequate efficiency.

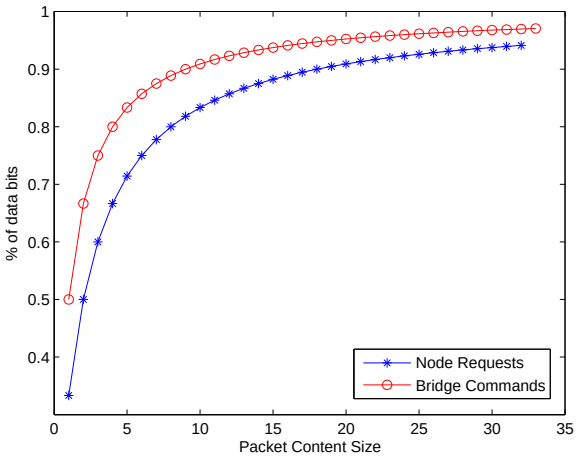

Figure 4.1: Uplink packet overhead.

In a similar manner, we can also illustrate downlink efficiency and packet overhead. Figure 4.2 can also be used to investigate efficiency of the downlink protocol. As it is mentioned earlier in this section, efficiency of node transactions is directly depends on the user and application. It can be optimized by using relevant messegaboxes effectively for read and write operations. Although efficiency of downlink communication is dependent on the user specifications, Figure 4.2 illustrates the performance enhancement with the usage of compressed translations and default read operations.

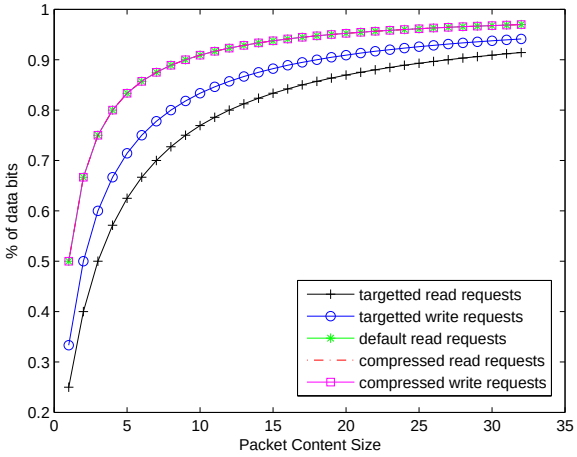

Figure 4.2: Downlink packet overhead.

Node transactions are usually periodic, therefore their effect on the performance of both downlink and uplink should be considered by users carefully. Besides, downlink heartbeat is also a periodic message defined strictly by the protocol. Downlink heartbeat is a write operation to inbox[0] in nature and handled by transmitting 3 bytes if targeted write is used or 2 bytes if compressed write operations are used. For both cases, heartbeat packets include 1 byte for command definition resulting 66.66% and 50% packet overhead respectively. This situation effects downlink efficiency for sure and should be used responsibly.

To sum up, it is obvious that network efficiency is dependent on the application and user definitions. However, the URB protocol provides some functionalities that improve performance of the bus. One of these improvements is the compressed and default read operations. These operations present a dense communication protocol handled both by transmitter and receivers. Second important improvement is the autonomous data transactions handled by bridges. So bandwidth occupation caused by periodic node transactions are shifted to downlink and only results of these transactions are directed through uplink connection. Synchronization messages can be considered as a bandwidth sacrifice for the sake of timely data acquisitions.

4.2.2 Media Access Overhead

Media access overhead is the network bandwidth used to arbitrate network access among transmitting nodes. Token passing approaches can be considered as examples to protocols that are exposed to media access overhead. However, the URB protocol is a deterministic approach to fieldbus applications. Each node is linked to a bridge, namely masters of the downlink, and can only transmit data with the request of a node. Responses to node request are also directly transmitted to the URB CPU by bridges. Therefore, the URB protocol is free from media access overheads.

4.3 Latency

Latency is the time delay between the moment a network transmission is initiated and the moment that takes effect within the destination. Therefore it is important for real-time network protocols to minimize communication latency. Latency properties of the URB protocol are constrained by many factors. This section expresses the experimental results over a USB system with RS232 uplink connection and I^2C downlink connection. The RS232 uplink raw bandwidth is set to 115200Kbit/s and I^2C downlink raw bandwidth is set to 1 MHz for these experiments.

4.3.1 Round-Trip Time

By definition, round-trip time (RTT) is the time required for a packet to travel from a specific source to a specific destination and traveling the same path back again. We used round-trip latency for the URB as the time required for the CPU to transmit a node request and receive the response for the specific package. Therefore, round-trip latency for the URB depends on the type of node transaction, size of response packet, and downlink transaction type as well.

Basic and the most common read transaction for every node application type is the **targeted** read node transactions. Series of targeted read transactions are caused by consecutive read operations from different outboxes. Figure 4.3 illustrates targeted read operations for 2 and 3 bytes of node data. Requesting 2 bytes of data and receiving the response packet took approximately 1.170 ms for the CPU with a variance of $7.79x10^{-5}$ ms. Similarly reading a 3 bytes of data results a round-trip time with mean value 1.284 ms and variance of $2.94x10^{-4}$ ms. It could be observed from these results that reading one extra byte from a node would increase the round-trip time of a read request by 0.113 ms.

The URB protocol implements default and compressed read operations in order to increase illustrated performance. While default read transactions are applicable to all kinds of nodes, compressed transactions should be supported by

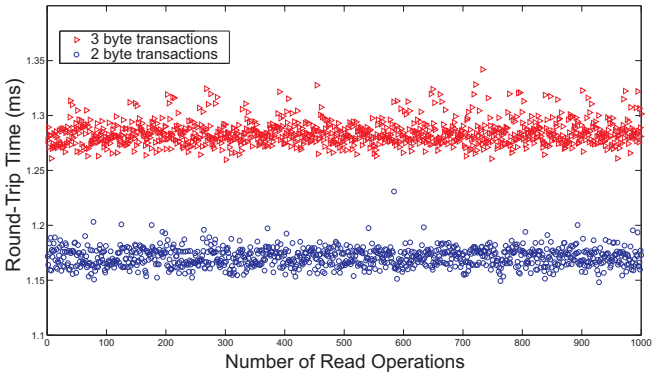

Figure 4.3: Round-trip time between the request and response of 1000 **targeted** read transactions from 2 and 3 byte long outboxes.

node hardware. (See Section 3.5 for details.) However, default read transactions are useful if a specific outbox data is requested periodically without requesting another data. Both compressed and default read transactions share the same notion, so resulting round-trip time values are approximately 1.139 ms and 1.135 respectively for 2 byte long data requests. Figures 4.4 and 4.5 illustrate test results for both transaction types from 2 byte and 3 byte long outboxes.

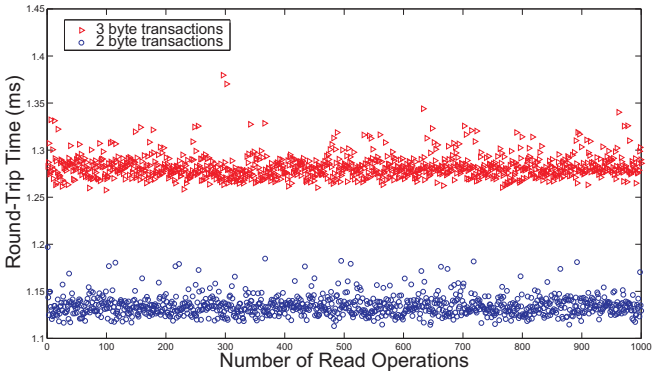

Figure 4.4: Round-trip time between the request and response of 1000 **default** read transactions from 2 and 3 byte long outboxes.

Finally, Figure 4.6 illustrates 2 byte long read operations for each transaction types and performance enhancement by using compressed and default read transactions. This slight improvement in latencies of read transactions would result significant enhancement in the performance of a system where periodic and frequent transactions are subject of matter. Besides this, Table 4.1 also summarizes

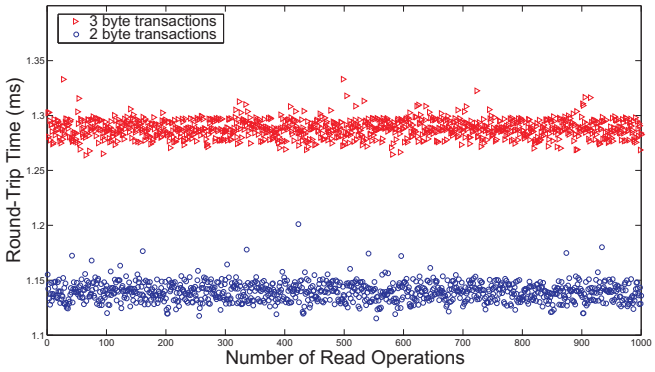

Figure 4.5: Round-trip time between the request and response of 1000 **compressed** read transactions from 2 and 3 byte long outboxes.

transaction round-trip latencies with mean and variance values.

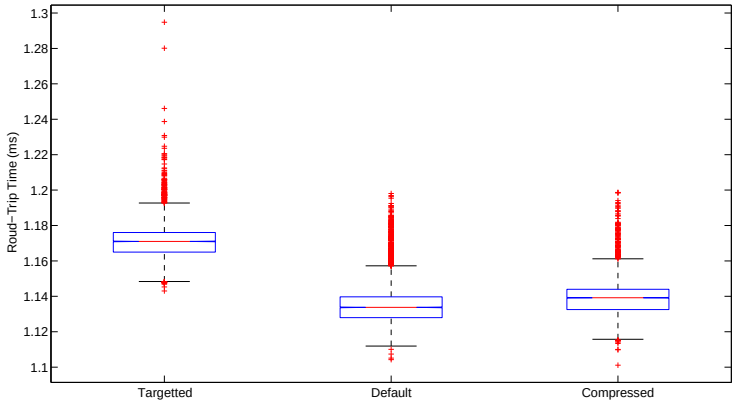

Figure 4.6: Mean and variance of round-trip times between the request and response of 10000 read transactions from 2 byte long outboxes for each transaction type.

4.3.2 Uplink Round-Trip Time

Section 4.3.1 gives the results for round-trip time measurements of a URB system. Round-trip time latencies of the whole URB system are actually caused by two components: uplink and downlink round-trip latencies.

By calculating uplink latencies we could also distinguish downlink from total

	Targetted		*Compressed*		*Default*	
	mean(ms)	**var**	**mean(ms)**	**var**	**mean(ms)**	**var**
2 byte	1.170	$7x10^{-5}$	1.139	$2x10^{-4}$	1.135	$2x10^{-4}$
3 byte	1.284	$2x10^{-4}$	1.286	$8x10^{-5}$	1.282	$1x10^{-4}$

Table 4.1: Round-Trip latencies for each kind of node request transactions.

round-trip delays. For instance, Figure 4.7 illustrates round-trip latencies of uplink **BRG_LED_CMD** command for 10000 requests. Each **BRG_LED_CMD** request is composed of 3 byte packets and results in 2 bytes of response packets. Experiments proved that uplink round-trip time for this bridge request is approximately 0.8846 ms with a variance of $6.0076x10^{-005}$ and standard deviation of 0.0078 ms.

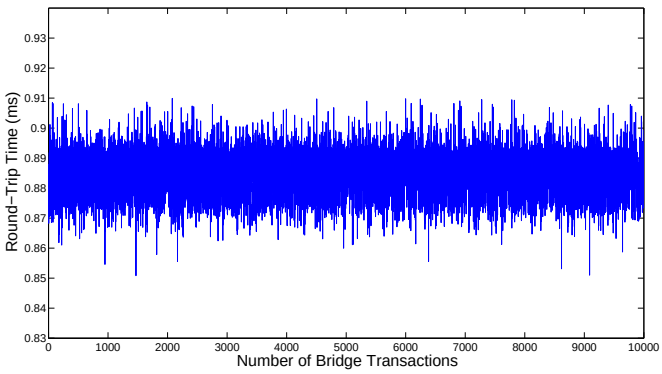

Figure 4.7: Round-trip time between the request and response of 10000 **targeted** `BRG_LED_CMDs`.

These results also proved that a great amount of round-trip time latency is caused by uplink subsystem. In other words, RS232 protocol creates a bottleneck that increases round-trip time latency. Being aware of this observation, we have proposed to implement uplink libraries for USB protocol. However, this thesis does not concern USB based uplink implementation.

4.4 Synchronization Performance

There are three different metrics that characterize the performance of URB synchronization algorithm. Firstly, **initial convergence delay** is the time delay

starting from when the microcontroller is powered up and successfully locks its local heartbeat to the downlink heartbeat signal. We performed our experiments on a SiLabs F340 board running at 48 MHz and the bridge generates a downlink heartbeat signal at every 1 ms. Resulting maximum initial delay is 88.4 ms and the average is 78.4 ms for a single node.

Second one of our experiments is performed to detect the **single-node jitter in** t_{adc_end} (See Figure 3.5). This metric is also important to leave a gap between t_{adc_end} and downlink heartbeat arrival time. We inserted a gap slightly larger than the maximum jitter after t_{adc_end} to guarantee that the downlink heartbeat always arrives after the end of data acquisition. This ensures that the sequence of events illustrated in Figure 3.5 is always achieved. Our experiments showed that maximum single-node jitter in t_{adc_end} is $1.33\mu s$ with an average value of $0.92\mu s$.

Last of all, we tested **inter-node difference in** t_{adc_end} in order to detect the difference between the end of data acquisition for individual nodes. Inter-node difference in t_{adc_end} actually illustrates the difference between local node heartbeats for different nodes. This metric is also important to verify the success of our synchronization algorithm and it points out how multiple nodes can successfully lock to the downlink heartbeat. Our experiments result in with a maximum of $0.56\mu s$ with an average of $0.19\mu s$. Table hede also summarizes these results.

Parameter	**Max**	**Average**
initial convergence delay	88.4 ms	78.4 ms
inter-node difference in t_{acq_end}	$0.56\mu s$	$0.19\mu s$
single-node jitter on t_{acq_end}	$1.33\mu s$	$0.92\mu s$

Table 4.2: Performance metrics for node synchronization. Results were obtained after 20 experiments measuring key transitions with an oscilloscope.

4.5 Determinacy

Determinacy is the ability to predict worst-case response time and it is very important for meeting real-time constraints of many embedded systems [48]. Most of the URB requests are started by the CPU and responded accordingly except

autonomous responses. Requests generated by the CPU are guaranteed to be finished within 2 ms and as it is already described, its variation is in degree of 10^{-4}. Therefore, CPU could calculate the worst-case delay of the response for a request.

However, autonomous requests are initiated periodically by bridges and responses are directed to the CPU. Determinacy for autonomous responses are also important in a system requires periodicity in data acquisition from distributed nodes. Assume that, CPU requests for autonomous responses from a two byte outbox with 10 ms period from a bridge. After enabling autonomous data fetch, bridge should response with the requested data for every 10 ms and the CPU should predict the reception time of the resulting message. Figure 4.8 illustrates this scenario by presenting periodicity of messages. These results verify that each autonomous response packet is received by the CPU within approximately 9.9370 ms with a standard deviation of 0.0069. As a result, an autonomously generated response packet is delivered to the CPU well before the CPU needs the data.

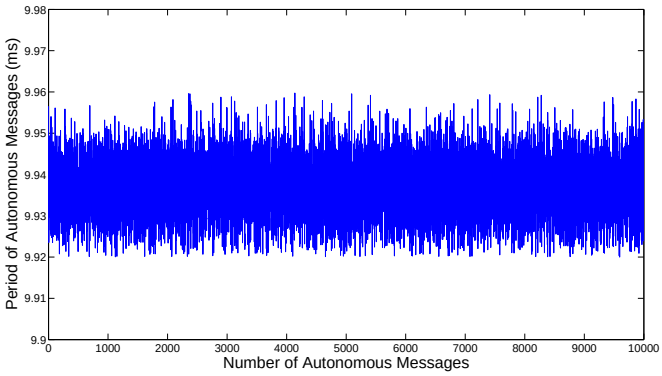

Figure 4.8: Period of 10000 autonomous responses generated for every 10 ms.

A prioritization mechanism is also included in some systems to improve determinacy of the system [49]. URB protocol also implements urgent messages for this purpose. However, we usually do not need urgent messages since each message is responded by nodes subsequent to its reception.

4.6 Robustness

Robustness is basicaly the ability to detect and recover from any error. Error detection is important for embedded systems and especially time critical systems like autonomous mobile robots[38]. However, most error detection techniques need significant amount of computation or cause bandwidth occupation. Therefore, the URB protocol left error detection and recovery from errors to the application layer and user of the URB.

One possible source of erroneous data transmission is channel noise, causing mis-decoding the data. The most popular solution of this problem is to use a Cyclic Redundancy Check (CRC). CRC is a function that takes any number of data stream as an input and it produces a a value with defined size. The resulting value is used as a checksum to detect alternation of data during transmission or storage as well. Therefore, memory errors caused by copying a stream of data within a node can also be detected by using CRC. A CRC function can be implemented by a user if the URB is used in a noisy environment that cause a great number of erroneous transmissions. It could be done easily by adding an extra byte to a message box as a checksum value.

Another erroneous situation can also be detected in the form of repeated messages. If a node fails to update a data acquired from a sensor or actuator, it would cause the CPU to read the same message located in the buffer. Some protocols include a serial number for each message to detect repeated or out of sequence messages. The serial number can also be used as a tool to detect error losses and it can be implemented by including extra bytes for serial number within a message box.

Although these solutions to detecting and recovering from errors increase the reliability of the URB system, they need either computational power or increase bandwidth occupation. Trade off between reliability and performance depends on the application type and constraints, so the URB protocol leaves this issue to the user.

Chapter 5

Evaluation and Conclusion

5.1 Evaluation

In most commercial network protocols, designers optimize the protocol to specific applications, resulting reduction in performance in different, incompatible domains [12]. For instance, the CAN is designed for reliable and deterministic communication within automotive networks. Firewire is also designed for fast isochronous data transmission without large receive and transmit buffers. On the other hand, LonTalk was tailored to provide flexibility. This section states the pitfalls of the most popular three commercial protocols and explains the reasons for designing and using the URB protocol.

5.1.1 Network Protocol Pitfalls

Although there are many communication protocols in embedded systems market, only very few of these protocols spread through the market[49]. LonTalk and CAN are accepted as embedded market standards, while Firewire is widely used for real-time isochronous communication. Therefore, we would discuss the performance of these three protocols in this section.

One of the most important drawbacks to LonTalk is its medium access control (MAC) protocol. As it is discussed earlier in Section 2.5.1, LonTalk implements a form of carrier sense multiple access with collision avoidance (CSMA/CA) algorithm for MAC protocol. However, CSMA/CA is a non-deterministic and probabilistic collision avoidance approach. Collision based protocols are not suitable for real-time applications, because randomness caused by the protocol makes it impossible to estimate message delivery time bounds. Nevertheless, collisions based protocols are not efficient in heavy traffic where nodes are synchronous. Each collision increases the number of nodes ready to transmit and it takes a long time to serve every waiting node. Lastly, LonTalk provides all six layers of OSI model implementation except application layer, which leaves users very limited access to protocol details and modification opportunity.

On the other hand CAN is one of the mostly used protocol for real-time applications. However, CAN specifies a basic set of network applications and leaves most of the application details to users. Opposed to LonTalk, implementation of a CAN system is costly and tricky with requirement to additional services. For instance, CAN protocol allows to transmit communication packets up to eight bytes and a fragmentation algorithm should be implemented by users to transmit packets larger than eight bytes. This algorithm should break data packets longer than eight bytes and reconstruct the data at the receiver side.

Firewire seems to be the most convenient commercial product for real-time robotic application with its support for isochronous communication. However fast communication speed of firewire requires the transmitted data to be handled as fast as communication speed. Otherwise accumulated data in receiving buffer would cause data overflow and data loss. Therefore micro-controller should be chosen wisely in order to prevent data loss which results increase in cost of the system. Besides, firewire requires packet headers larger than 20 bytes which occupies most of the bandwidth in a system only requires transmission of small and periodic data packets.

5.1.2 Advantages of the URB

Taking into consideration above pitfalls of most popular commercial networks, the URB presents an efficient, fast, real-time, synchronous and cheap communication infrastructure for internal communication within mobile robots. Master-slave architecture gives the full control of the bus to the URB CPU, hence collisions and contentions between nodes are controlled and handled by a central authority. Unlike probabilistic collision avoidance algorithms, the URB protocol is completely deterministic and matches well for real-time applications. Deterministic protocol applied by the URB is also gives satisfactory results within synchronous heavy traffic.

On the other hand, supported libraries and APIs provide enough flexibility and modularity for users while abstracting details of the communication protocol. Complex protocols like LON limits user flexibility and protocols like CAN requires a longer implementation phase. Being aware of disadvantages of both edges, we implemented a more user friendly and modular development library supporting various commercial microcontrollers.

Most of the commercial microcontrollers presents support for popular I^2C and RS232 communication mediums. This gives users the flexibility to use desired microcontroller with computational power suitable for a specific application. Besides, one can minimize the system cost by choosing the right microcontrollers for the system. Therefore, URB is not as expensive as Firewire protocol but providing satisfactory embedded communication performance.

5.2 Conclusion

In this thesis, we described a new modular, robust and real-time communication protocol for small and mid-sized mobile robots. The Universal Robot Bus protocol stands for the communication infrastructure interfacing local sensor and actuator nodes to a central processing unit. Our application domain is composed

of robots requiring modular, extensible and reliable systems for performing real-time tasks robustly. Therefore, URB presents a reliable, modular and extensible communication architecture with real-time communication capabilities.

One of the most important features of the URB framework is the automation of periodic data transfer requests. The other important functionality presented by the URB is the automatic synchronization of data acquisition across multiple nodes. Both of these features are implemented and analyzed throughout this thesis. They help reducing the communication workload over the central prosessor and improving data acquisition reliability, respectively.

The first URB architecture was implemented on 8051 microcontrollers developed by Silicon Laboratories (SiLabs) by using UART (RS232) and SMBUS (I^2C) modules of these chips. Current CPU libraries and drivers provide support for RS232 and USB on Linux platforms. However, our intended goal is to port these libraries and drivers into QNX operating system in order to use URB within dynamic hexapod robot platform called *RHex*.

Bibliography

[1] *Universal Serial Bus Specification*, April 27 2000.

[2] E. V. Appleton. The automatic synchronization of triode oscillators. In *Proc. Camb. Phil. Soc.*, volume 21, pages 231–248, 1923.

[3] ARCNET User Group e. V. *ARCNET*.

[4] Aricent. *Auto-Discovery Approaches*, 2006.

[5] K. Arvind. Probabilistic clock synchronization in distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 5(5):474–487, May 1994.

[6] . T. Association. 1394 standards and specifications summary. Technical report.

[7] H. D. Bellescize. La reception synchrone. In *L'onde electrique*, volume 11, pages 225–240, 1932.

[8] R. E. Best. *Phase Locked Loops-Design, Simulation and Applications*. McGraw-Hill, fourth edition edition, 2003.

[9] S. T. Bushby. Bacnet: A standard communication infrastructure for intelligent buildings. *Automation in Construction*, 6:529–540, 1997.

[10] Contemporary Controls. *ARCNET Tutorial*.

[11] E. Corp. *Introduction to the LONWORKS System*, 078-0183-01a edition.

[12] A. Dean and B. Upender. Embedded communication network pitfalls. *Embedded Systems Programming*, 10(9), 1997.

[13] J. Encinas. *Phase Locked Loops*. Springer, 1993.

[14] J. L. Fernandez, M. J. Souto, D. P. Losada, R. Sanz, and E. Paz. Communication framework for sensor-actuator data in mobile robots. In *Proceedings of the IEEE International Symposium on Industrial Electronics*, pages 1502–1507. IEEE, June 2007.

[15] C. Fetzer and F. Cristian. Lower bounds for convergence function based clock synchronization. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 137–143, New York, NY, USA, 1995. ACM.

[16] M. Gergeleit and H. Streich. Implementing a distributed high-resolution real-time clock using the can-bus. In *Proceedings of the International CAN Conference*, Mainz, Germany, 1994.

[17] S. Heath. Echelon-networking control. *IEE Review*, 38:363 – 367, 1992.

[18] S. Hong and W.-H. Kim. Bandwidth allocation scheme in can protocol. *Control Theory and Applications, IEE Proceedings* -, 147(1):37–44, Jan 2000.

[19] T. Hsu, B. Shieh, and C. Lee. An all-digital phase-locked loop (adpll)-based clock recovery circuit. *IEEE Journal of Solid-State Circuits*, 34(8):1063–1073, 1999.

[20] R. Huntoon and A. Weiss. Synchronization of oscillators. *Proceedings of the IRE*, 35(12):1415–1423, December 1947.

[21] T. Instruments. Application report: Comparing bus solutions, March 2000.

[22] J. Janet, W. Wiseman, R. Michelli, A. Walker, M. Wysochanski, and R. Hamlin. Applications of control networks in distributed robotic systems. *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, 4:3365–3370 vol.4, 11-14 Oct 1998.

[23] H. Kopetz. Event-triggered versus time-triggered real-time systems. In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, pages 87–101, London, UK, 1991. Springer-Verlag.

[24] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, 2001.

[25] H. Kopetz and G. Grünsteidl. Ttp-a protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, 1994.

[26] H. Kopetz and R. Nossal. Temporal firewalls in large distributed real-time systems. *Distributed Computing Systems, 1997., Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of*, pages 310–315, Oct 1997.

[27] Q. Li and D. Rus. Global clock synchronization in sensor networks. *IEEE Transactions on Computers*, 55(2):214–226, 2006.

[28] D. Mills. Internet time synchronization: the network time protocol. *Communications, IEEE Transactions on*, 39(10):1482–1493, Oct 1991.

[29] M. Mock and E. Nett. Real-time communication in autonomous robot systems. *Autonomous Decentralized Systems, 1999. Integration of Heterogeneous Systems. Proceedings. The Fourth International Symposium on*, pages 34–41, 1999.

[30] U. Nunes, J. A. Fonseca, L. Almeida, R. Araújo, and R. Maia. Using distributed systems in real-time control of autonomous vehicles. *Robotica*, 21(3):271–281, 2003.

[31] S. PalChaudhuri, A. K. Saha, and D. B. Johnson. Adaptive clock synchronization in sensor networks. In *IPSN '04: Proceedings of the third international symposium on Information processing in sensor networks*, pages 340–348, New York, NY, USA, 2004. ACM.

[32] R. Patzke. Fieldbus basics. *Computer Standards and Interfaces*, 19(5):275–293, 1998.

[33] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[34] P. Perez, J.-L. Posadas, G. Benet, F. Blanes, and J. Simo. An intelligent sensor architecture for mobile robots. In *Proceedings of the Int. Conference on Advanced Robotics*, pages 1148–1153. IEEE, 2003.

[35] Philips Semiconductors. *The I2C-BUS specification*, 2.1 edition, January 2000.

[36] P. Ramanathan, K. G. Shin, and R. W. Butler. Fault-tolerant clock synchronization in distributed systems. *Computer*, 23:33 – 42, 1990.

[37] Robert Bosch GmbH, Postfach 30 02 40, D-70442 Stuttgart. *CAN Specification*, 2.0 edition, 1991.

[38] L. Rollins. Embedded communication. Technical report, Carnegie Mellon University, Dependable Embedded Systems, 1999.

[39] K. Römer. Time synchronization in ad hoc networks. In *MobiHoc '01: Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, pages 173–182, New York, NY, USA, 2001. ACM.

[40] F. B. Schneider. Understanding protocols for byzantine clock synchronization. Technical report, Ithaca, NY, USA, 1987.

[41] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[42] D. Semiconductors. Application note 83: Fundamentals of rs232 serial communications, 1998.

[43] D. E. Simon. *An Embedded Software Primer*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[44] R. Subramanian and I. D. Scherson. An analysis of diffusive load-balancing. In *SPAA '94: Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, pages 220–225, New York, NY, USA, 1994. ACM.

[45] K. Tindell and A. Burns. Guaranteed message latencies for distributed safety critical hard real-time networks, 1994.

[46] C. Travis. Automatic frequency control. *Proceedings of the IRE*, 23(10):1125–1141, October 1935.

[47] P. W. M. Tsang and R. W. C. Wang. Development of a distributive lighting control system using local operating network. *IEEE Transactions on Consumer Electronics*, 40:879 – 889, 1994.

[48] B. P. Upender and P. J. Koopman. Embedded communication protocol options. In *Proceedings of Embedded Systems Conference*, pages 469 – 480, October 1993.

[49] B. P. Upender and J. Philip J. Koopman. Communication protocols for embedded systems. *Embedded Systems Programming*, 7(11):46–58, 1994.

[50] P. Verissimo and L. Rodrigues. A posteriori agreement for fault-tolerant clock synchronization on broadcast networks. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 527–536, Boston, MA, July 1992.

[51] P. Verissimo, L. Rodrigues, and A. Casimiro. Cesiumspray: a precise and accurate global time servicefor large-scale systems. *Real-Time Syst.*, 12(3):243–294, 1997.

[52] J. H. Vincent. On some experiments in which two neighboring maintained oscillatory circuits affect a resonating circuit. In *Proceedings Royal Society*, volume 32, pages 84–91, 1920.

[53] M. Wargui and A. Rachid. Application of controller area network to mobile robots. In *Proceedings of the Electrotechnical Conference*, volume 1, pages 205–207, Bari, Italy, May 1996.

Appendix A

Bridge Request Commands

Table A.1: Bridge command types with description

Bridge Command Type	**#args**	**#resp**
BRG_RESET_CMD	0	N/A
Reset the bridge and all nodes. *Arguments*: NONE *Response*: NONE even if RR=1		
BRG_GETVER_CMD	0	2
Request bridge firmware version and revision. *Arguments*: NONE *Response*: Version followed by revision (each in one byte).		
BRG_CLOCK_CMD	0	4
Request current clock value of bridge. *Arguments*: NONE *Response*: 4 byte integer with most significant byte first.		

Continued on next page.

Bridge Command Type	**#args**	**#resp**
BRG_LED_CMD	1	1
Sets the states of a specified LED located on each bridge. *Arguments*: 1 byte to define the LED number (MS 7 bits) and next state of the specified LED (LS 1 bit). *Response*: 1 byte to define the previous state of the LED.		
BRG_PKTCOUNT_CMD	0	8
Request uplink and downlink packet counts. *Arguments*: NONE. *Response*: 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. *Arguments*: NONE *Response*: 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. *Arguments*: 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. *Response*: 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. Continued on next page.		

Bridge Command Type	**#args**	**#resp**
Arguments: 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. *Response*: 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. *Arguments*: 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. *Response*: 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. *Arguments*: 1 byte with node address. *Response*: 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. *Arguments*: NONE *Response*: 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. *Arguments*: NONE *Response*: 0xFFA5B99F (MSB first).		

Continued on next page.

Bridge Command Type	**#args**	**#resp**
BRG_ECHO_CMD	1	-
Directly echoes the argument through uplink. *Arguments*: 1 byte to be echoed. *Response*: 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. *Arguments*: NONE *Response*: 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. *Arguments*: Variable *Response*: Variable		
BRG_UL_GETVER_CMD	0	10
In response to this command, uplink version, revision and type string is sent. *Arguments*: NONE *Response*: 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. *Arguments*: Variable *Response*: Variable		

Appendix B

Code

Sample node firmware implementation

```
#include "silabs_util.h"
#include "test_urb_node.h"
#include "urb_node_lib.h"

byte xdata ADC_Value[4];
byte xdata buttonValue[4];
byte *tempPtr;

/**< Period between successive calls to app_update().
Units are in microseconds */
#define URB_NL_UPDATE_PERIOD 4000

void SYSCLK_init( void ) {
  unsigned char delay = 100;
  //set internal oscilator - divided by 1 (highest)
  OSCICN |= 0x03;
  CLKMUL = 0x00;        // Reset the clock multiplier
  CLKMUL |= rbMULEN;    // Enable clock multiplier
  while ( delay-- );      // Delay for >5us
  // Initialize the clock multiplier
```

```
  CLKMUL |= rbMULEN + rbMULINIT ;
  // Wait for multiplier to lock
  while( !(CLKMUL & rbMULRDY) );
  // Select 4X multiplier as the system clock
  CLKSEL = (CLKSEL & ~rmCLKSL_MASK) | rvCLKSL_4X; }

void PORT_init( void ) {
  P0MDOUT = 0x00;    //P0 pins are open-drain
  P0 = 0xFF;
  P2MDOUT |= 0x0C;  //P2.2 and P2.3 are push-pull
  P2 &= ~0x0C;
  // P2.5 is analog input for analog acquisition
  P2MDIN &= ~0x20;
  P2SKIP |= 0x20;
  XBR0      = 0x04;
  XBR1      = 0x40; }

void ADC0_init ( void ) {
  // normal tracking mode; ADC0 conversions are initiated
  // with AD0BUSY; ADC0 data is left-justified
  ADC0CN = 0x00;
  AMX0P  = 0x04;  // P2.5 analog input source
  AMX0N  = 0x1F;  // ground as negative input
  ADC0CF = 0x80;  // ADC conversion clock = SYSCLK/16
  ADC0CN |= 0x80; // Enable ADC0
  EIE1 |= 0x08;  /* enable ADC interrupts */ }

void T2_init ( void ) {
  // Configure and initialize Timer 2.
  // Use SYSCLK/12 as timebase
  TMR2CN  = 0x00;
  // Timer2 clocked based on T2XCLK;
  CKCON  &= ~0x60;
  // Init reload values and set to reload immediately
  TMR2RL  = -URB_NL_UPDATE_PERIOD;
```

```
  TMR2    = TMR2RL;
  // Enable Timer2 interrupts and start Timer 2
  ET2     = 1;
  TR2     = 1; }

void app_bootup( void ) {
  PCA0MD &= ~0x40;     //disable watchdog timer
  SYSCLK_init();
  PORT_init();
  ADC0_init();
  T2_init();
  URB_SetClass( 1 );
  URB_SetIndex( 10 );
  URB_SetVersion( 10 );
  URB_SetRevision( 3 );
  URB_SetAddress( 14 );
  URB_SetupOutbox( 0x01, 2, ADC_Value );
  URB_SetupInbox( 0x02, 2, buttonValue, true );
  LED = 0;
  EA = 1; /* enable global interrupts */ }

void app_reset( void ) {
   byte i = 0;
   for ( i = 0; i < sizeof( ADC_Value ); i++ )
      ADC_Value[i] = 0;
   for ( i = 0; i < sizeof( buttonValue ); i++ )
      ADC_Value[i] = 0; }

void app_init( void ) {
   EIE1 |= 0x08;    // enable ADC interrupts
   tempPtr = XNULL; }

void app_idle( void ) {
  if ( URB_CheckInbox(2) ) {
     byte *tmp = URB_LockInbox(2);
```

```
    if (*tmp) LED = 1;
    else      LED = 0;

    if (*(++tmp)) LED2 = 1;
    else          LED2 = 0;

    URB_ReleaseInbox(2);
  }
}

void app_update( void ) {
  AD0BUSY = 1; }

void app_signal( byte arg ) {
  if ( arg == 0xff )
    LED2 = ~LED2;  }

void app_fault( byte errno ) {
  byte tmp = errno; }

void AD0_ISR (void) interrupt INTERRUPT_AD0_EOC {
   AD0INT = 0;
   tempPtr = URB_LockOutbox( 1 );
   *tempPtr = AD0H;
   tempPtr++;
   *tempPtr = AD0L;
   URB_ReleaseOutbox( 1 );
   URB_UpdateDone(); }
```