Lightweight Wired Communication for Sensor and Actuator Arrays Using I^{2}C Bus

by

Vaccaro Alessandro, B.Sc.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment of the requirements for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2011

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Vaccaro Alessandro

August 31, 2011

Permission to Lend and/or Copy

Acknowledgments

Many thanks to my supervisor, Jonathan Dukes, for all the relentless support and very valuable suggestions he gave me throughout the work.

Many, many, thanks to my whole family. My parents Marina and Giannantonio, for they infinite persuasion to never give up.

I want to remember all the love I have for brothers and sisters, Chiara, Zeno, Carlo Alberto, Carlotta, Giovanni ed Edoardo.

VACCARO ALESSANDRO

University of Dublin, Trinity College

September 2011

Lightweight Wired Communication for Sensor and Actuator Arrays Using I^{2}C Bus

Vaccaro Alessandro, M.Sc.
University of Dublin, Trinity College, 2011

Supervisor: Dukes Jonathan

Wireless communication is very popular nowadays, however a wired medium could be ideal in those situations where the use of wireless communication is not practical due to economical or environmental limitations. This dissertation proposes to use the two-wire I^{2}C bus to implement wired sensor and actuator networks. A new software layer, to run on micro-controllers, is introduced to extend the capabilities of I^{2}C and provide the basis for communication in an array of sensors or actuators. This software extension is suitable for generic embedded systems communication.

An I^{2}C hardware controller is commonly integrated in many commercially available micro-controllers, making it a good candidate in a wired sensor or actuator network. To make it suitable for this use, I^{2}C has to be expanded, overcoming some of its limitations. The maximum number of devices supported on a single bus is 128 (or 1024 using a newer addressing schema, not available on every MCU). Each device's address is statically assigned, before to be plugged on the bus. The proposed software layer will overcome this limitation extending the address space and introducing dynamic address assignment. The wire's electrical capacity is another factor limiting the number of devices per bus. The protocol will be compatible with I^2C Multiplexers which allow to handle several channels at time. A channel is an independent I^2C bus. The software extension transparently handles data transfer from one channel to the other and the channel access arbitration. All channels are joint into a single virtual bus.

The I^2C bus is meant for on PCB communication, where there is no need for device hot-swapping. However to use I^2C as medium for sensor array nodes, the bus has to handle hot-swapping. To achieve this feature the proposed protocol improves the bus access arbitration and introduces a distributed address assignment routine. The software extension is meant only for micro-controllers, however it is backward compatible with standard I^2C devices, on which it is impossible to load any kind of software.

This dissertation describes the design and the implementation of the proposed software extension. Such prototype is composed of a software stack and the hardware on which to run it. The proof of concept is based on the AVR ATmega328 micro-controller. The evaluation of the protocol, based on such prototype, has demonstrated how effective it is to overcome the aforementioned I^2C bus limitations.

Contents

Chapter 1

Introduction

Many research project are focusing on Wireless sensor networks. The possibility to interconnect several small embedded devices without any physical infrastructure has a certain appeal to many monitoring applications. Several challenges have to be addressed yet. For example how to maintain an ad-hoc network and at the same time allow the nodes to switch off the radio to save power [4]. Power consumption is a primary problem which may also lead to communication reliability issues. In some context communication faults are no option. Also the possible interference of radio waves might be a problem, particularly in those environment where already few wireless networks have to coexists. Wireless Sensors Networks (WSN) lead the designers of an hypothetical system to make a tradeoff among computation, power efficiency and communication reliability. This dissertation proposes a wired alternative to WSN (and maybe in future actuator networks), which tries to maintain, where possible, the flexibility of a WSN node, but at the same time overcome any data transfer fault or power limitations (to a certain extent).

Such software extension is based on I^2C standard specification [10], a serial bus for embedded systems, introduced in 1980 by Philips. It enables to connect a number of devices in a serial manner, with low power requirements and small protocol processing overhead. Each device has to have a predefined static address before to be connected to the bus, and at most 1024 devices can be connected to the bus. As aforementioned the alternative will try to be as flexible as possible. One challenge is indeed to overcome the fixed and pre-programmed address limitation of I^{2}C to allow to plug a new devices without an address predefined. The problem here is to arbitrate the access to the bus. Also the medium length is a key issue. I^{2}C wire length is limited by the maximum electrical capacity of the whole system, which can not be over 400μF. The electrical capacitance raises proportionally to the number of devices connected and the length of the wire. Such problem has one electrical root that can be addressed partially, via software, by reducing the transfer speed. The system should be able to sense the electrical capacitance of the wire and inversely proportional adjust the communication speed. Many more technical constraints raises in this project: are all outlined in the next few sections.

Previously were explained the motivations and the technical issues of the system. However its use in the real world are possibly various. Few use cases have been envisioned, without any ambition of being actually feasible. A scenario where to use such bus might be a super market digital price labeling system. Although centrally managed wireless options are already available a wired electronic labeling system might take advantage of the shelf to host the wires. The electronic labels this way do not have to deal with any power issue. Also the physical structure of the scaffoldings embeds the wire, making a more efficient use of the available resources. Price labels are moved only when the related product is moved. However most of the time each product stays in the same location, making a wireless price tag an unjustified cost.

A road tunnel monitoring system may be build using the I^{2}C Extender. Sensors could be plugged on the bus inside a road tunnel, giving environmental data, such as temperature, pollution and vehicular traffic levels. The reduced energy requirements make possible to run the system on small backup batteries. Such system could give prompt notice of an incident and precise information about where it happened. Faulty sensor nodes can be easily replaced, due to the dynamic addressing feature. It might be possible an active use of the bus, not only to sense environment data, but also to light up lights placed on the road marks. This would make an active signaling system to inform drivers of sudden queues, incidents, water overflows or any other hazardous road condition.

Precision farming is another context that might take advantage from the proposed I^{2}C Extender. Soil moisture and acidity sensors might be embedded in the water pipes alongside with the bus wires. The data collected by these sensors could be used to remotely control the irrigation valves, also installed on the water pipes. Such kind of system would optimize drip irrigation, reducing water and fertilizers waste. A wired system would not suffer any radio communication decrease during crop flourishing, as precision farming researchers have already experienced [7]. Many more application can be envisioned, for those context where the installation of a wireless infrastructure is not feasible or uneconomical.
There are three main areas the I^{2}C Extender should implement features for. The main point is dynamic address assignment. Previously was explained how applications can benefit by such feature. Having an automatic assignment of address slot, would make possible the hot plugging of new nodes to the network, without any address clash. One of the nodes on the I^{2}C network will play the role of System Host, to coordinate the operations on the bus. Each device will have to ask the system host to be assigned a valid (available and not reserved) address, picked from the list it will maintain. Also the number of nodes is a limit which has physical implications, due to the added electrical capacity of each added node. The standard I^{2}C allows two different addressing schema, 7 and 10 bits, allowing at most 128 and 1024 devices connected at the same time. The actual address space is smaller, indeed there are eight addresses reserved for bus management purposes. The I^{2}C Extender proposes to enlarge the address space, using 16-bit addresses. The electrical capacity issue will be partially solved splitting the bus via commercially available I^{2}C bus multiplexers. The following, are the main requirements of the I^{2}C Extender, each one is accompanied by a short description of the problems to address.

- Automatic addressing has to be implemented to facilitate hot-plugging of new devices onto an I^{2}C bus. The protocol is not intended to deal with the electrical implication of hot-pluging but will only provide a set of instructions to allow a Client to negotiate an address with the System Host. The assigned address remains constant during the device online time. A requirement of the I^{2}C Extender is to avoid any pre-existing hardware address. Other protocols, such as SMBus, rely on hardware addresses. Address retention to overcome temporary power losses is an interesting feature that will be considered for implementation.

- The address space (using the standard 7-bit is 2^7, for 127 addresses) will be increased implementing a 16-bit Client ID, allowing more than 64K addresses. This addressing schema works parallel to the standard 7-bit I^{2}C address. The old address space is retained for compatibility with older devices, and will be used to subdivide the nodes in clusters.

- To better handle the increased number of devices is introduced a Multicast feature. The I^{2}C Extender will support up to 64 (2^6) Multicast groups. Any Client device registered in any of these groups will receive the packets the System Host sends to the group.

- The electrical capacitance of the bus is the main limit to the number of Clients on the bus. This problem is overcame subdividing the bus into channels using off the shelf components such as the NXP PCA9544 bus multiplexers. Addressing devices on different channels will happen transparently. The channels will appear as a single virtual bus by time-sharing the System Host to each channel in Round Robin.

- All these features will be implemented on standard commercially available hardware, with no need for extra components, excepted the I^{2}C multiplexer. This requirement will ease the implementation of the I^{2}C Extender on a wider set of systems.

I^{2}C bus is simple and widely used either in consumer and industrial electronics. There are a wide variety of existing devices using the I^{2}C bus, ranging from real time clocks, EEPROM and Flash memories, Analog to Digital or Digital to Analog converters just to cite some. The I^{2}C bus needs only two wires plus shared ground and two pins on each Integrated Circuit, as shown in Picture 1.1. Its simple design and low footprint makes it cost effective reducing complexity and wiring. Therefore is suitable to interconnect any kind of device that needs to communicate with sensors or actuators where hardware complexity is undesirable.

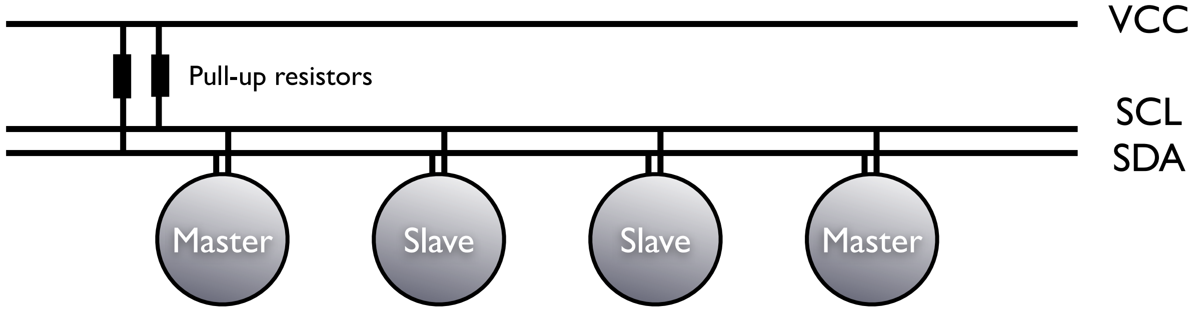


Figure 1.1: A diagram showing the devices involved in the system. A system host node, few smart slaves and a regular slave (e.g. a real time clock).

I^{2}C is generally used for on board communication, however it makes sense to try to stretch the bus over the standard specifications to interconnect arrays of sensors over long distances. Implementation of the I^{2}C bus on long wires (up to 1 Km) is possible, as demonstrated in [8], the goal of this dissertation is to get a reliable connection on a 100 meters long wire. The use of such a technology to build array of sensor where even a hundred of units must share the same bus needs the introduction of a dynamic address resolution makes possible to hot swap devices enforcing a clash free environment. Such address assignment system will have to deal with a large number of devices all trying to access the medium at the same time. In a scenario where all the nodes have been placed on the bus, as soon the system is powered, all the nodes will try to contact the master, to get an address. It is a key requirement for the bus to ensure a strong bus access arbitration, to avoid more than one device trying to write on the bus. The current bus arbitration provided by the standard I^{2}C specification will be described and all the limitations highlighted.

1.1 Dissertation Contributions

In the previous chapter have been illustrated some constraints of the I^{2}C bus. Also were brought here examples of applications that could benefit from a improvement of te I^{2}C . Briefly these issues of the bus are:

- Poor flexibility of devices' static addresses.
- The maximum number of elements in the bus might not always be sufficient.
- Transparent bus interconnection. Several independent busses (Multiplexer channels) might be transparently connected to share data.
- Medium length - wire electrical capacity. The communication speed should change in a inverse proportional manner.

Some of these features are already present in other interconnection busses. However not all of them are available on the same technology, or that bus poses limitations of other nature. After having analyzed how other protocols implement their services, such as the auto address slot allocation.

In particular will be proposed a protocol that will extend the functionalities of the bus I^{2}C , adding auto addressing of the devices, long distance communication, channel multiplexing. Also will be experimented a feature to locate the position of each node, relatively to the bus master (as introduced in "Master-Slave message propagation timing" section). The implementation of said software layer will be carried on per steps. Starting with the auto address assignment. The work procedure will be design and test. If something should fail then another design or approach will be tested. The next step will be the introduction of Multicasting, useful to synchronize a group of clients, or other purposes. Then this software layer will be integrated with existing hardware multiplexers, such as the PCF9544 which enables the subdivision of a bus. The System Host will poll the clients on that segment, to receive or issue commands from or to the various Clients. Since each channel is de facto a independent I^2C bus, the electrical capacity of the whole system can be split over 4 or more channels. Each channel is independent by the others and has a full address space available. Therefore each channel could potentially address all the 64K Client IDs of the new addressing schema. However this configuration is hardly viable.

Communication over long distances is another important feature. The proposed I^2C Extender should be compatible with the hardware I^2C buffers, available on the market and designed to reduce the overall electrical capacity of the system. Also a technique to gauge the capacity of the system should be sought. This is a necessary step to know how much the speed has to be reduced to keep a reliable communication.
The plan is to implement the system host on a NXP ARM7 μController, while each slave device will run on ATmega AVR328. The ARM7 implementation will use FreeRTOS, a real-time operating system for embedded computers [1], while on the AVR328 will run BeRTOS [2] another small real time embedded operating system.

1.1.1 Auto-address Configuration

Bus arbitration is the biggest issue the protocol has to cope with. Having so many (is expected an address space of 65,495 Client IDs over several interconnected buses), trying to acquire a new address from the master is a big issue. The master has no knowledge about the nodes connected on the bus, so its duty of each single Client to ask for an address. As soon as the bus is initiated the Master will broadcast its own address to everyone on the bus. Afterwards the devices will try to acquire the bus to contact the Master. This massive competition for the medium has to be coped dealing with the

[1]FreeRTOS, http://www.freertos.org
[2]BeRTOS, http://www.bertos.org

small number of lines of the bus (there are only two lines available). A novel approach is developed trying to keep it as simple as possible to limit the overhead. Computational power is a serious constraint, due to the hardware characteristic of the Client devices.

1.1.2 Address Space and Multicast

The standard 7-Bit address space is increased to 16-Bit. The new address schema is not supposed to be a complete replacement for the I^{2}C addressing, which is kept and used by the I^{2}C Extender. Indeed the new address space, in the form of a 16-Bit long Client ID, makes use of the old 7-Bit address to group the Clients into clusters. These clusters are balanced and automatically populated by the System Host. It is a necessary feature to make the I^{2}C Extender possible. The new address space also introduces the concept of multicast groups which are 64 of the new 65,536 Client IDs. These are logical groups used to group the Clients by common characteristics. These groups can be created or deleted at run time and Clients can be registered into them at any time. A Client can exist without being registered in any group. This design keeps the I^{2}C Extender backward compatible with the standard I^{2}C devices.

1.2 Structure of the Dissertation

- Chapter 2 will cover the state of the current bus technologies. These will be briefly described, showing their points of strength and also their weakness. For each bus is explained why it can not be adopted as a base to construct the protocol hereby proposed.

- Chapter 3 is a detailed description of the I^{2}C bus. Is very important to show what the bus can actually do, and what can not. In the chapter is explained how some of its features are used by the I^{2}C extension. However the description is brief, and the details are left in the following chapters.

- Chapter 4 contains the full specification of the proposed software extension. Each aspect of the protocol is carefully covered and the various design decisions justified. The main parts are "Auto Addressing" and "Bus Channel Multiplexing". The first explains how the devices agree on each other address, without causing clashes and allowing backward compatibility with old I^{2}C devices. In the latter section focus on how the protocol "joins" tow distinct busses using an hardware multiplexer.

- Chapter 5 is about the implementation of a prototype. Either the software and hardware parts are described.

- Chapter 6 contains the protocol evaluation based on the prototype described in Chapter 5. It also contains the conclusion of the dissertation.

Chapter 2

State of the Art

The State of the Art chapter covers the current situation of the bus technology. Only the most relevant protocols are investigated in this section. The requisites for the bus were wiring simplicity, small cpu overhead, implementation simplicity. Of the suitable protocols have been analyzed their hot swapping, number of hosts per bus, multi master configuration and backward compatibility features. Each selected bus is described in detail and the points of strength outlined. To finish is explained why these busses do not match entirely the specification of the I^{2}C extension. Such specification includes: limited hardware modification, no prefixed hardware IDs, software only solution, multi-master configuration, very high number of devices per bus line/system host and transparent interaction with a multi-plexed bus. Also this protocol should be able to reduce the communication speed, to make it able to work in a situation were the electrical capacity is high (for I^{2}C this means very close to the specified maximum which is 400pF [8, 10]).

2.1 Applications of Low Power Buses

As mentioned in the introduction widely used in various contexts. Few possible scenario for the software extension to I^{2}C were outlined. Their main characteristic is to be very ambitious, deployed on large scale environment, where the long distance communication is a key feature. Also the dynamic address allocation is essential in those scenarios, where the wire was proposed instead of wireless network. Indeed the features brought by this extension of the I^2C bus, might make feasible the actual implementation of those systems. However at the moment bus technologies such as I^2C or SMBus are deployed in a more orthodox manner, and far less ambitious. Following there are five examples of how these buses are nowadays used, and how they might benefit from the features of the hereby described extension. Fields of application ranges from modular robotics, expandable wireless sensor nodes and large scale tactile sensors. These papers do not specifically focus on the role of the I^2C or the SMBus, but still can give an interesting overview of their application field.

The first application is a snake robot able to crawl on the ground and swim [1], is mechanically designed to be waterproof. It is designed for an outdoor use, in particular for the inspection of pipes. One of the most interesting characteristic lies in its locomotion, based on the output of a pattern generator, which makes use of stabilized rhythmic patterns. Such approach makes the pattern very robust to external perturbations. The snake is a modular robot. Several segments can be connected together and the current electronic makes possible to join up to 127 modules, which communicate via I^2C bus. Each segment is completely independent to the others, having its own battery pack, a motor controller and its own DC engine. Each motor controller is based on a PIC16F μController. The paper has a very detailed description of the power and logical electronics, which goes out the scope of this dissertation. However points out the fact the patter generator is run in a μController in the head. The head works also as I^2C master and sends pattern instructions to all the segments using the aforementioned bus. On the robot is run the standard implementation of the I^2C bus [1]. Due to its "distributed" and modular structure the robot would benefit from the dynamic address allocation of the nodes proposed in this dissertation. Further segments could be simply plugged into the robot, without the need of reprogramming the head node. There would be no benefit from the extension of the maximum number of nodes applicable to the bus, due to the nature of the robot.

In [6] is described an expandable robot architecture, which uses sensors and actuators, each of them is a module managed by micro-controller. It proposes a device distributed approach to realize an expandable robot system. The idea is to have a modular robot, where components can be plugged and utilized with no change to the hardware and only a minimal software update. Software update is necessary: in case a new accelerometer is added the system still need to know how to use its data. Each device is "smart", having its own μController. The key aspect of this design is to be distributed, and each device, once receive a command, can execute it independently by the main CPU. It is interesting to note how the amount of wiring needed to connect all the modules is dramatically reduced, compared to a centralized design. Each module is connected to the main cpu through the SMBus. SMBus is the chosen technology because it permits dynamic plug of new modules (SMBus supports auto addressing) using only two pins on the module's micro-controller. SMBus is also good because its mostly compatible with the standard I^2C . However this design still have some flaws. As stated by the researcher, the problem rely on SMBus itself [6]. Although it is ideal due to the dynamic address allocation, on the other hand is limited to 127 devices because it uses the 7 bit schema. Also has much stricter voltage and timing limitations. SMBus is a multi master bus, so it fits the design of a device distributed robot. On the other hand it permits only one active node per time, dramatically affecting the response time of the whole system. All this problems are addressed by the I^2C software extension proposed in this dissertation. It would be ideal because it allows dynamic address slot assignment, and permit to split the bus in several sub networks via the usual I^2C multiplexer. These subnetworks can act as independent buses, though being still able to communicate with the other sub networks, using the system host as router. Using the I^2C software extension it would be possible to have independent bus lines for each limb of the robot, increasing the response time of each section. While all the accelerometers and gyroscopes, could sit on the trunk of the robot, on their own bus line, where the main μController can poll them as frequently as needed, without interfering with the movement actuators.

Tactile sensors are getting ever more important in robotics. A robot able to sense the pressure can be programmed to hold and manipulate very fragile objects. In paper [12] is described a conformable tactile sensor surface. Is a network of matrix of tiny pressure sensor elements. The surface (made of urethane) is soft and flexible. The sensing area can be adjusted by increasing or reducing the tactile elements (the small pressure sensor matrix). The actual pressure sensor element is a photo-relector, covered by the urethane foil, which provide mechanical protection from impactive forces. Each Photo-reflector consumes around 50mA, which can add to 50A for a skin with 1000 sensing spots. This is a huge amount of current. To overcome this problem the system implements a sort of time sharing, so the whole skin is scanned per ares. This solution also reduces the number of analog-digital converters. Such digital skin is wired with a combination of SMBus and an ad-hoc ring LAN is used to address a tactile element which can have up to 65536 sensing spots.

This number is very high and close to the aim of this dissertation work [1]. However the I^2C extended bus, here proposed reaches similar levels using only I^2C and avoiding the need to mix heterogeneous bus technologies. The digital skin sensor is sub-netting the bus using the said ring network. Such solution need a central node which switches from one "tactile master" to the other, in order to access the various sensing foil, such tactile master has to interface itself with the LAN, therefore introducing a delay. The software extension for the I^2C bus hereby proposed, would help to reach the same number of sensors, using one single technology.

Another application of the I^2C is shown in [5] where an expandable wireless sensor network node is proposed. Each node can be extended by piling several boards. All the extension boards can communicate using the I^2C bus. In this case there is not too much convenience in using the I^2C software extension. The SMBus alone would be flexible enough, since it allows automatic address slot assignment.

[1]The devices here involved are standard slaves.

In [16] is discussed a wearable computing experiment. This is indeed an interesting application where the I^{2}C bus is necessary. The garment was covered with some magnets were sensors could be attached. The magnets also work as connectors for serial data, clock, and power. Sensors are connected the I^{2}C bus which was chosen due to its simplicity, the low power it needs to function and the fact is implemented by a vast amount of commercial devices. Here, as mentioned for [6], the software extension for I^{2}C would make possible to have a separate bus for each limb. As stated in the paper the main problem the systems has to deal with is conflicting addresses, indeed such application is the ideal testbed for the I^{2}C software extension. It provides a solution to every issue, from dynamic node swapping, to the synchronization and data routing of a sub-neted bus. The presence of several nodes, connected with unmasked wires increases the electrical capacity of the whole system. Such situation is ideal to test if the I^{2}C extension can alter the clock speed to the varying set up of the bus (more nodes implicates lower speed).
This short walkthrough of research papers has demonstrated how the I^{2}C bus would benefit of the feature hereby proposed. The possible application fields are vast and different, but all have in common the need for more flexibility in node hot swapping and scalability.

2.2 Low power wired bus architectures

Here few words to introduce the section.

2.2.1 SMBus

The SMBus is based and compatible with I^{2}C , is the main inspiration for the proposed protocol. It allows hot swapping of unpowered devices on the serial bus, and overcome any address clash by providing a system to auto allocate address slots. Each SMBus device has a unique 128-bit long hardware address, used to request the regular 7 bit long I^{2}C address. SMBus is less flexible than I^{2}C , from an electrical and timing point of view. Indeed all the nodes have to run at the same voltage, and the clock frequency is strictly defined. SMBus is a multi-master bus, as well as I^{2}C . Following are listed the most interesting features of the bus, and some comments where these are too limited for an extended usage.

- ARP Master: is a master node entitled of assigning addresses to the slave devices. In many cases the ARP Master and the Bus Master are coincident. Only one ARP Master per bus is allowed [3].

- Assigned addresses: legal values for a slave device address are between 0010 000b and 1111 110b (only 7 bit addressing allowed). This means that no more than 110 devices can be plugged on the bus at the same time [3]. The ARP master node is a sort of system host, to which all the node refer to, when, among the other features, in need of an I^{2}C address.

- Fixed Slave Address: is a device not able to get a new address by the ARP Master. The master in this case should not assign a slave an address used by a non-ARP capable device. This feature is necessary to keep compatibility with the I^{2}C devices [3]. The SMBus system host node keeps a list of all the "non smart" nodes address, and will never assign one of these to a new node.

- Persistent Slave Address: In case of power loss the address is retained by the device. This feature is very interesting, and useful, since the most complex and computation intensive phase of auto addressing is the bus arbitration. Address retention is necessary to avoid redundant requests to the master node and keep busy the bus [3] after a power loss or a general hardware reset.

- Used Address Pool: a list of slave address known to be used by non-ARP capable devices, assigned to slave devices, reserved (like the default address, used to broadcast messages from the master to all the other nodes). A smart slave should never be assigned one of these addresses [3].

- UDID, Unique Device Identifier: each device (ARP capable) must have one of these id, that will be used only during the ARP phase. SMBus defines it as a 128-bit long address [3], necessary to the ARP Master to address the right node when assigning a new 7bit address. This requirement of the SMBus might lead to problems covered in the next lines.

2.2.2 1-Wire Bus

It's a one wire bus solution designed by Dallas Semiconductors. On a single wire this bus can provide a communication channel, signaling and power.

Power is granted on the slave device, by a small capacitor, which store power when the line is high, to release it when it goes down, see picture 2.1. As soon the stored power in the capacitor is drained out, the device will enter a reset state and will not reply to the queries anymore. This scenario might happen during a long sequence of transmitted zeros, when there is little chance for the parasite powering circuit to recharge [15].

Due to the absence of a clock line, the transaction from a logical zero to a logical one (and vice versa) is based on strict timing, to which either the master and the slave have to comply with. Every bit is transmitted over a timeframe of 60μs. The slot always start with a logical '1', the bus line is retained down for less than 15μs. To write a '0' the line has to be kept down for the entire length of the time slot, 60μs. There are exception to this rules, and a time slot can be extended to 120μs [14].

Signaling is also provided through the single wire. To send a reset line signal the bus is kept down for 480μs.

The bus is therefore very simple, and is indicated to communicate with relatively simple devices, such as small EEPROM memory, temperature sensors, realtime clocks. The bus is single master based, hence only one master device is allowed, also due to the simplicity of each device. This preclude any "smart" behavior from the slaves. However there are some very interesting features on this bus that will be analyzed in the next

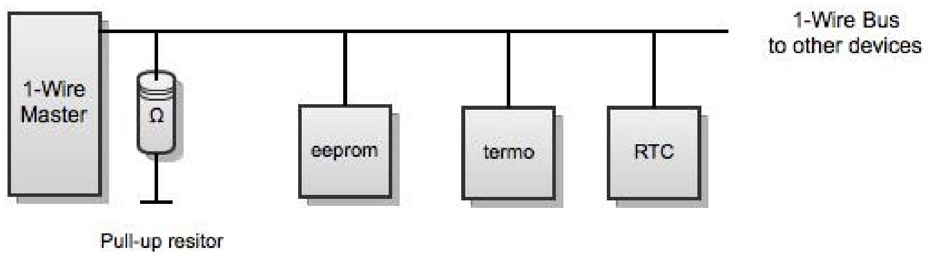


Figure 2.1: The wiring diagram of the 1-Wire bus. The connected devices are "scavenging" the power from the data line, which is high when idle or transmitting a logic 1. Each node store power in an internal small capacitor.

chapters that might be a good source of inspiration for the implementation of the proposed I^{2}C extension. This feature is the SEARCH ROM command, an algorithm that allows the master to make an inventory of any non smart device connected on the bus.

2.3 Similarities in SMBus and 1-Wire

2.3.1 SMBus

With SMBus version 2 was introduced the ARP function, which lets to auto assign address slots to the devices on the bus. Address assignment uses the standard AND-Wired arbitration rules of the I^{2}C . Once assigned an address remain constant for all the time the device is powered up. Also address retention over a power loss period is allowed. Every assigned device address is then used as any other I^{2}C 7 bit address, and does not require further overhead.

The system host (ARP Master) always execute the ARP service either when it enters a working state or when it receives a bus state change signal. The process starts by initializing the Used Address Pool, populated at the beginning only by slave fixed addresses. Then the master will issue the command "Prepare to ARP", as in flow chart 5.4. The diagram comes from the official SMBus specification [3]. If any acknowledgement is received then the ARP master will send the "Gey UDID" command. It will wait for answers by the slave. If it get no answer then it assumes the device is no longer active. Otherwise takes the address sent by the slave. If it is 0xFF then select a valid not used address and assigns it to the devices witht he command "Assign Address".

If the device has a valid address then the ARP master tries to figure out if the device is a fixed address by comparing the bits 126 and 127 of the UDID, if these bits are 00 then the device is a fixed address. This address is added to the fixes address pool if not present. If the said address belongs to a "dynamic device" then its address is checked against the Master Address list. If is not used then the node can keep its address, otherwise a valid one will be selected and assigned to that node.

2.3.2 1-Wire

The 1-Wire protocol does not define any auto addressing schema, since each device is univocally identified by a factory programmed 64 bit long address, which guarantees an almost infinite number of addresses, making senseless any address resolution algorithm. However the protocol provides a very handy command, called SEARCH ROM [14], that allows the master node to discover which devices are plugged on the bus, without having to know their address. This feature is very useful to the purpose of keeping backward compatibility with already existing I^{2}C hardware. SEARCH ROM uses a binary tree search algorithm, to discover in a relatively short period of time the devices connected. The implementation exploits some feature of the 1-Wire bus that make this technique very fast. The I^{2}C bus implementation will not be as fast as on the 1-Wire, but is still worth to be implemented. This feature is covered in deatail in section 3.3.

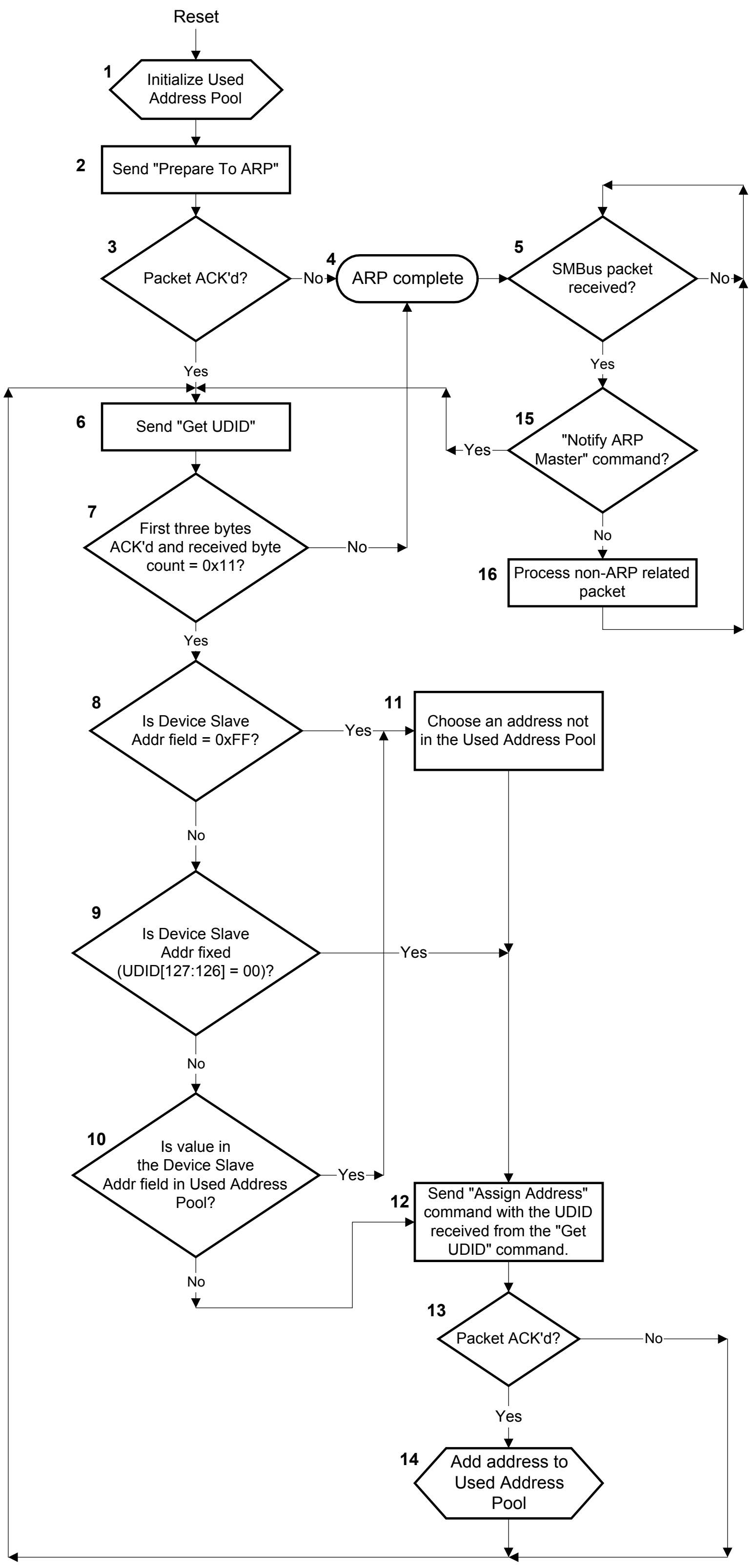

Reset
1
Initialize Used Address Pool
2
Send "Prepare To ARP"
3
Packet ACK'd?
No
4
ARP complete
5
SMBus packet received?
No
Yes
Yes
6
Send "Get UDID"
15
"Notify ARP Master" command?
Yes
No
7
First three bytes ACK'd and received byte count = 0x11?
No
16
Process non-ARP related packet
Yes
8
Is Device Slave Addr field = 0xFF?
Yes
11
Choose an address not in the Used Address Pool
No
9
Is Device Slave Addr fixed (UDID[127:126] = 00)?
Yes
No
10
Is value in the Device Slave Addr field in Used Address Pool?
Yes
12
Send "Assign Address" command with the UDID received from the "Get UDID" command.
No
13
Packet ACK'd?
No
Yes
14
Add address to Used Address Pool

Figure 2.2: The flow chart of the ARP feature of the SMBus version 2.0, this flow chart is quoted from the official specification of the bus [3].

Chapter 3

I^{2}C Bus

This chapter will examine address auto allocation as done by other protocols meant to be used in an embedded systems context. The two which shares more similarities with I^{2}C are SMBus and 1-Wire. Although SMBus is very similar to the hereby prosed protocol, will be shown how SMBus is not fully satisfactory solutions for the issues the I^{2}C extension aim to address. 1-Wire bus, is more minimal than I^{2}C , requiring only 1 wire for data, signaling and power. Is not suitable as a communication medium for complex embedded devices, due to the lack of multi-master capabilities. The most relevant features of these protocols will be discussed and compared against the requirement of the I^{2}C extension.

3.1 I^{2}C Bus Features and Characteristics

I^{2}C is an serial bus designed by Philips and released to the market in 1982. Originally was used for IC communication inside TV, its main use is for IC communication on the same PCB. It runs at different frequency: 100kbit/s in the Standard-mode, 400kbit/s in Fast-mode, 1Mbit/s in Fast-modePlus up to 3.4Mbit/s in the High-speed mode. Fast IC can reduce their frequency to work on buses mixed with low speed elements. I^{2}C is a multi-master bus and only needs two wire, thus reducing the size of chip and the wiring cost. These wires are SCL (Serial Clock) and SDA (Serial Data). The two lines are

AND-Wired, meaning that if two masters are writing at the same time, the actual value visible on the bus is the logical and of the two values. Such idea is the base of the I^{2}C bus arbitration, necessary in case of multi-master system. There is only one active master per time. Other masters will behave as slave. Each device has its own hardware address, which is static. Most of the IC have n pins available to set the last n digits of the address, thus enabling more devices of the same kind to coexist on the same bus.

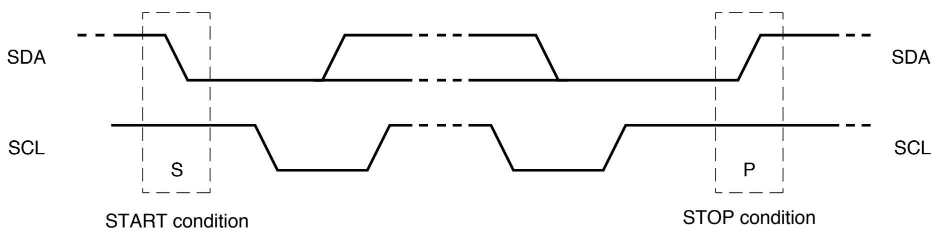
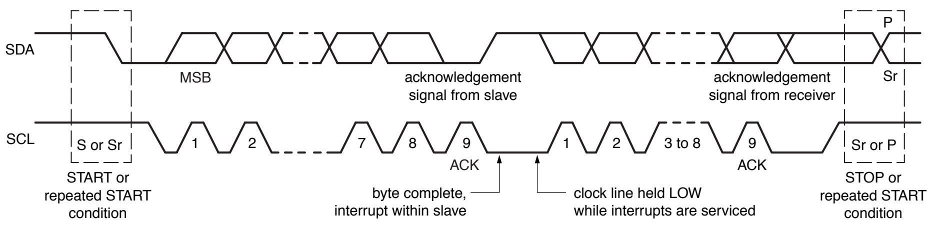


Figure 3.1: Start and Stop conditions, to acquire the bus and to release it. This timing diagram is quoted from the I^{2}C standard specification [10].

As shown in 3.1 to acquire the bus and start the communication cycle the SDA line must be pulled down while the SCL is high. Similarly the STOP condition happen if the SDA line is pulled up when the SCL line is high. These two conditions are always triggered by the master. Once the Start condition is issued, the bus is considered busy. The I^{2}C bus is byte oriented. The minimum data chunk sent is indeed a byte which is always acknowledged by the receiver.

The communication always start with a START condition which is followed by a 7 bit slave address plus the Read/Write bit. After the actual data is transferred, and



SDA
SCL
P
MSB
acknowledgement signal from slave
acknowledgement signal from receiver
Sr
S or Sr
1
2
7
8
9
ACK
1
2
3 to 8
9
ACK
Sr or P
START or repeated START condition
byte complete, interrupt within slave
clock line held LOW while interrupts are serviced
STOP or repeated START condition

Figure 3.2: The Byte and ACK is minimum communication unit, and is always acknowledged. This timing diagram is quoted from the I^2C standard specification [10].

acknowledged every byte. The session ends when the master place a STOP condition on the bus. At this point the bus is considered available. 10 bit addressing is also available. This was a feature introduced after the first specification, and had to keep its compatibility with the existing ICs. So a reserved address is used (11110XX) but the last two digits are the first two digit of the actual 10 bit address, whose other 8 bits follows in the next byte, as shown in 3.3. The older 7 bit ICs will ignore this call, because that address is a reserved one.

If a master needs to issue several commands to a slave, for example first to read and then to write, it can do a repeated start condition to avoid to loose the bus. This is a very simple feature of the I^2C bus. Once the first cycle of transfer is completed, the master reissue another START condition, instead of the STOP. This way it keeps the right on the bus. The full sequence of byte transfer is shown in picture 3.4

I^2C allows a master to send a general call, a command that is received by all the node. Is a sort of broadcast in the IEEE 802.3 networks. If a slave does not need the data issued

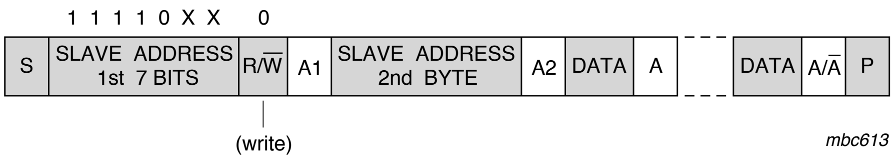


Figure 3.3: To address a 10 bit slave is used the normal 7 bit addressing but is used a reserved address, for the first 5 bits, while the last 2 are the first 2 bits of the actual 10 bit address. This timing diagram is quoted from the I^{2}C standard specification [10].

can ignore the command, and not acknowledge the command. All the other slaves will acknowledge, but the master node is not aware of how many nodes are responding. The second and n+1 bytes will be acknowledge by all the slaves able to handle that data. The general call address is a byte all at 0. The general call has many function, most of the time different for each IC. In some cases it is useful for a node to identify itself on the bus as by picture 3.5.

3.2 I^{2}C in Multi-Master Configuration

In the earlier sections the players of the I^{2}C bus have introduced. These are a Master node and a Slave node. The Master is always the node which initializes the transaction. A Master can be in transmitting and receiving mode (up to the least significant bit of the address byte). I^{2}C bus supports more than one Master on the same bus, therefore a Master might happen to be a Slave. Is also possible that two or more Masters initiate a

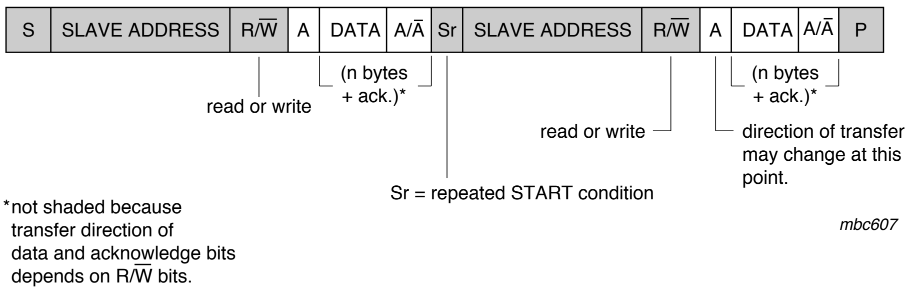

S
SLAVE ADDRESS
R/$\overline{W}$
A
DATA
A/$\overline{A}$
Sr
SLAVE ADDRESS
R/$\overline{W}$
A
DATA
A/$\overline{A}$
P
read or write
(n bytes + ack.)*
Sr = repeated START condition
read or write
(n bytes + ack.)*
direction of transfer may change at this point.
*not shaded because transfer direction of data and acknowledge bits depends on R/$\overline{W}$ bits.
mbc607

Figure 3.4: Repeated START condition to keep the right on the bus. This timing diagram is quoted from the I^{2}C standard specification [10].

transaction at the same time, but the communication would result completely garbled. Therefore I^{2}C has a method to decide which Master is allowed to continue the transaction. This section is about the techniques implemented by I^{2}C to arbitrate the access on the medium.

3.2.1 Bus Arbitration

As aforementioned in a multi-master configuration more than one master might try to access the bus at the same time, leading to potential transmission conflicts. So a bus arbitration process has to happen among the master initiating a transaction. Slave devices are never involved in the process, indeed this part of the I^{2}C protocol is not needed in a single master configuration.

In case two masters attempt to write a START condition on the bus, is necessary to determine which master acquire the bus and the right to carry on the transaction. This process is called arbitration and happens bit by bit and each time the SCL is HIGH

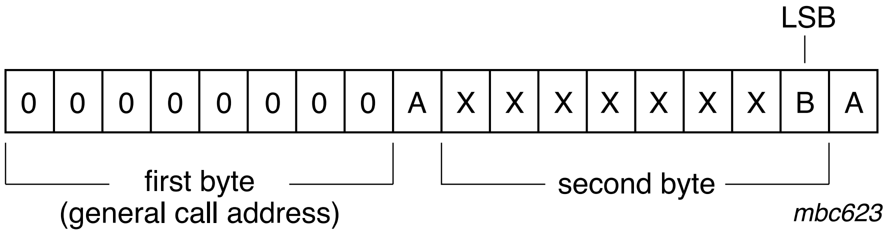


Figure 3.5: A general call transfer sequence, to note the least significative bit of the second byte, also used by an hardware master to advertise itself on the bus. This timing diagram is quoted from the I^{2}C standard specification [10].

(therefore no changes on the SDA are allowed) each master will compare the value on the bus with what it has wrote. Such a process can go on for several bits. It is also possible two master are able to complete a full communication without resulting into errors, given the data written is the same. The arbitration happens when one master reads a different value by the one it has written. For example, if it tries to write an HIGH on the SDA, but it reads LOW, it means some other master has pulled down the SDA line. This master then knows it has lost the arbitration, stops writing and might turn into slave mode. The other master instead will complete its transaction.

The arbitration is possible due to the AND-Wiring of the SDA driver of the master and the actual SDA line. If someone tries to place an HIGH on the bus while another places a LOW, the result will be LOW, as in the logical AND, where 1 and 0 gives 0. Also the SDL line is AND-Wired to support other features as in subsections 3.2.3 and 3.2.2.

The primary research focus of this dissertation is to push the limits of the I^{2}C bus. I^{2}C AND-Wired arbitration might not be the best option to cope with the overcrowded bus

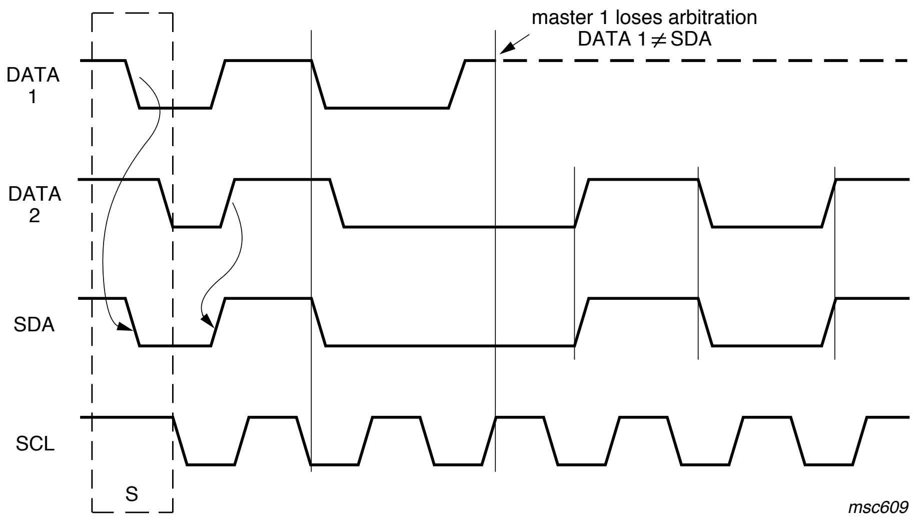


Figure 3.6: The arbitration process takes works by AND-Wiring the data controller of the master with the actual SDA line. If the bit written on the data controller is different by what actually on the bus, it means the master has lost the arbitration. This timing diagram is quoted from the I^{2}C standard specification [10].

envisioned in this document. The new protocol requires a fully multi master environment, so the number of elements trying to acquire write access to the bus is very high. This paper will first experiment the behavior of the current bus arbitration policy in presence of 15 masters on the same bus. The recorded results will be then compared to a bus arbitration technique very similar to a back off based access policy. When a master looses the arbitration it will back off for a random time before to transmit again. This should reduce the chances that after several masters have lost arbitration they try to write altogether, leading to another conflicting situation. Another optimization is to make the arbitration happen as soon as possible, with the lowest number of bytes transmitted. Transmitting a random first byte might increase the chance of arbitration, since it is done on bit by bit basis.

3.2.2 Clock Synchronization

Arbitration can happen either at Data level as well as at clock level. Different masters can have a different clock. I^{2}C has a schema to synchronize the serial clocks from all masters. Having all the same clock frequency will facilitate the arbitration process seen in subsection 3.2.1. When a master pulls LOW the SCL line, if this stays LOW after the LOW period has expired means that another master is keeping it LOW. The first master keeps counting off the time the SCL line stays LOW until it reaches an HIGH state. All the masters will start the timer for the HIGH state. The first master whose timer timeout will pull LOW the SCL line. So the clock is synchronized among all the masters to the longest LOW period and to the shortest HIGH period.

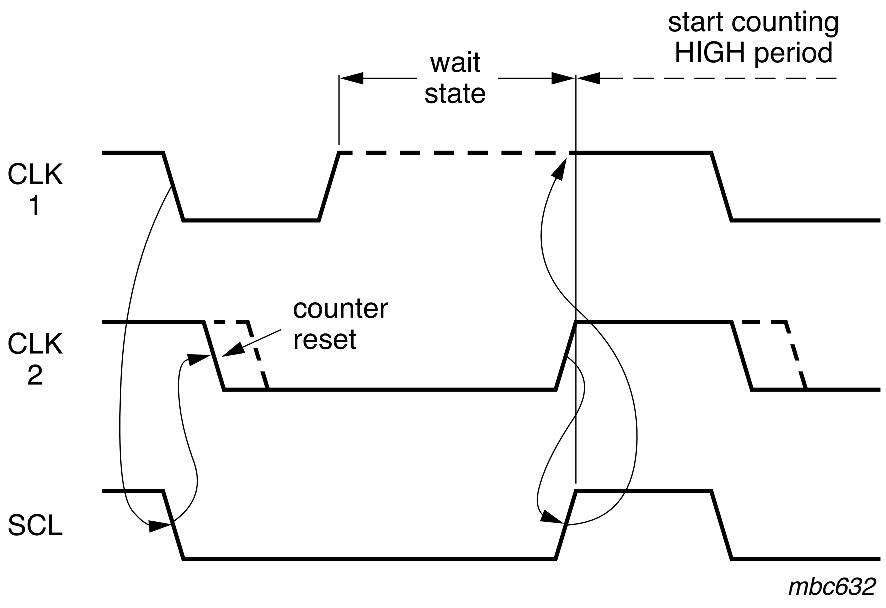


Figure 3.7: This diagram shows the clock synchronization process. Each master adopts the longest LOW period and the shortest HIGH period. All the masters will go through the transmission with the same clock frequency. This timing diagram is quoted from the I^{2}C standard specification [10].

3.2.3 Clock Stretching

The clock stretching is a feature performed by the receiving slave. It allows the slave to hold LOW the clock line forcing the master transmitter in a wait state until the line is driven HIGH again. Such a feature might be very useful in those cases a slave has to process the bytes as soon it receives them. It might take longer than an I^{2}C clock cycle, so it can pause the master from transmitting holding the line LOW. As soon the slave has completed its processing it turns HIGH the clock and the master transmitter can transmit the following byte. This feature is optional, and most I^{2}C slave devices do not implement it. However it can be very useful in the context of the I^{2}C Extender where all the Clients are micro-controllers and they can take advantage from this option. However any Client should never stretch the clock during a General Call or a multicast transaction. The implementer of the I^{2}C Extender protocol should take care to disable this feature when this particular exceptions happens.

3.3 Compatibility With Existing ICs

Existing I^{2}C hardware don't provide any auto-discovery function. Therefore the master has to be programmed with hard coded the addresses of these devices. This does not let the hot swapping of new devices on the bus. The following section will explain how a binary tree search is used to automatically discover this kind of devices connected on the bus. I^{2}C does not provide any command to discover what kind of service a node provides, making impossible the use of existing hardware once it is discovered. However this is still a very useful function, to avoid to assign an address that is already assigned to another device, making backward compatibility possible.

This feature is implemented similarly to the 1-Wire SEARCH ROM command, picture 3.8 shows the flow chart of this auto discovery function. The Search Rom function id is F0H, so its logical flow will start from the conditional block containing that ID. The Search Rom function is the most complex of the ROM function of the 1-Wire Bus. However ti can still be executed in few milliseconds, the standard specification show how it can run in only 13 milliseconds [14]. An approach similar to this might be implemented over the I^{2}C bus. Would be very helpful to the system host to auto-discover non smart devices plugged on the bus. Therefore allowing the systems host to mark their addresses as occupied.

3.4 Conclusion

In Chapter 3 were covered the main features of the I^{2}C bus. Its simplicity and yet ability to support sophisticated features make it the best candidate for the hereby proposed protocol. This chapter has covered the most intimate mechanisms of the I^{2}C protocol, necessary to understand how the I^{2}C Extender will fit flawlessly on it. Features such as the General Call and the various synchronization procedures are necessary to extend the I^{2}C protocol itself. Some I^{2}C Extender features, such as the multicast addressing will make extensive use of the General Call, trying at the same to keep the system overhead at the lowest level.

In Chapter 4 there is an extensive description of the approaches considered in the I^{2}C extension to comply with the requirements. The next chapter will focus on the design of the protocol of the I^{2}C Extender, introducing the reader to the new mechanisms designed to fulfill the requirements.

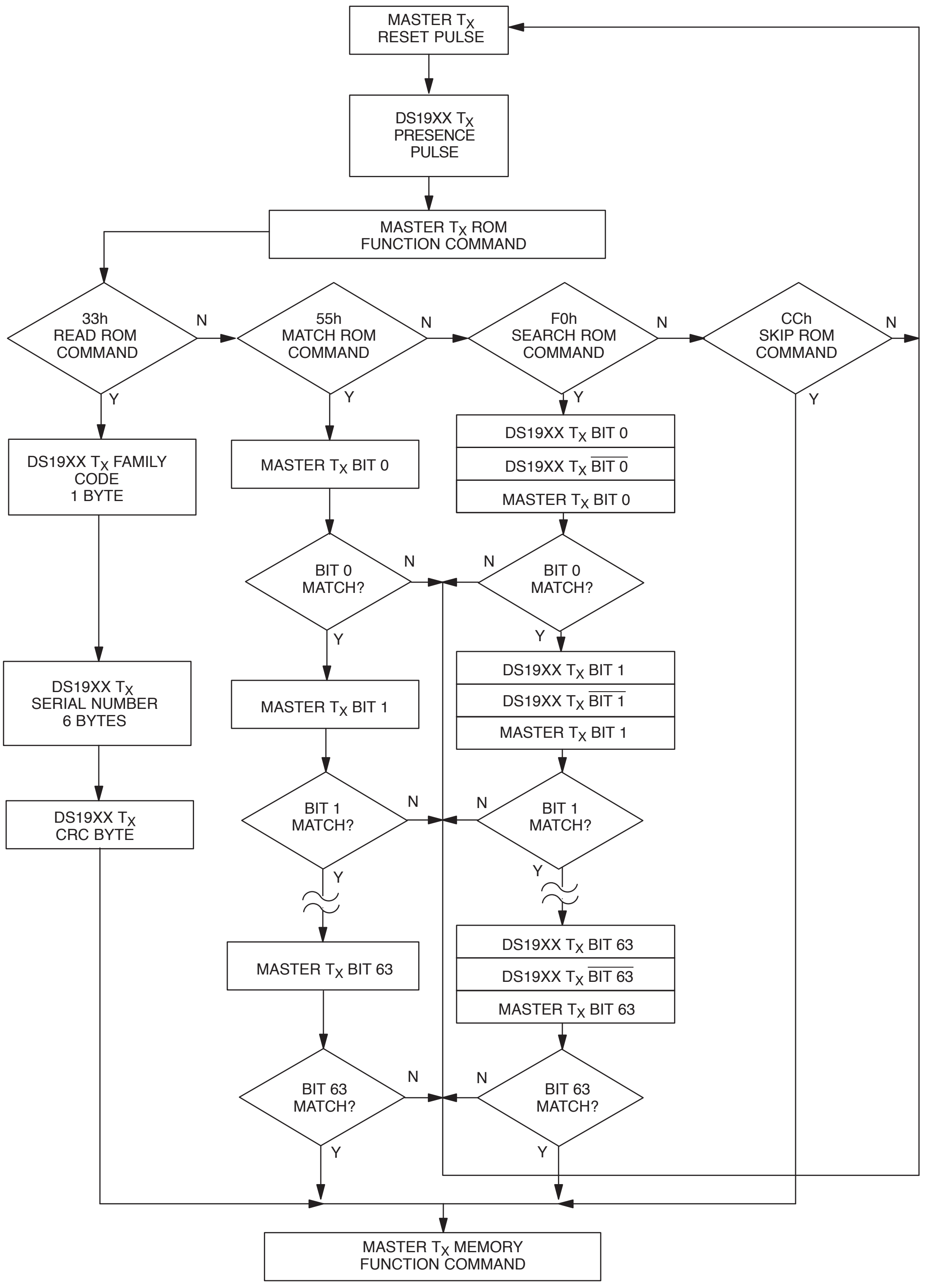


Figure 3.8: The flow chart of the SEARCH ROM function of the 1-Wire bus. It allows to discover in few milliseconds which devices are plugged on the bus. This graph is quoted from the 1-Wire specification document [14].

Chapter 4

Protocol Design

This chapter is meant as a specification document of the protocol. It shows the communication logic, what are the differences among the Client side and the System Host side and how they interact to each other. The aspects of the I^{2}C Extender covered in this chapter are built on the standard I^{2}C specification seen in the previous chapter. Although for every improvement brought by the I^{2}C Extender will be revised the relative standard behavior, is better to read the Chapter “I^{2}C Bus” before this one. This chapter is generic, and does not cover any implementation detail. It explains how the addressing schema changes to allow features like a wider address space, node clustering and multicast. Every aspect is explained at high level, to allow to replicate the same characteristic on the most various micro-controller typologies.

Each design choice is presented, discussed and evaluated in the chapter. Later in the chapter, is shown why each transaction is kept atomic and why the protocol is packet oriented. Before to close the chapter a section will introduce some improvements to the protocol, or alternative design choice not implemented in the final specification.

The last section of the chapter, "Improvements and Alternative Design", describes issues that might arise in particular operational states. Certain hardware configuration might create problems partially avoidable via software. In that section are proposed possible solutions. Along these are described alternative approaches to some features of the I^2C Extender. The section is intentionally cursory, meant as a source of suggestions for future improvements.

4.1 Addressing

As seen in the earlier chapter 2, the SMBus protcol, based on I^2C , allows hot-swapping. However it needs a pre-defined, unique 128 bit hardware id. Which could be seen as a sort of hardware modification and since each id also identify the producer, it reduces the spectrum of hardware compatible with the bus. However the main point of the I^2C extension is exactly to avoid any hardware changes, keeping it as standard as possible. Also the SMBus protocol is not implemented in many general purpose micro-controllers, making it no option to construct sensor and actuator arrays. Another reason to not use SMBus is the limited number of nodes attachable on the bus, capped to 128 (120 due to some of the address space reserved for some protocol features). The I^2C extension changes completely the addressing schema of I^2C , increasing dramatically the address space available. However the protocol does not deal with the physical limit, imposed by the electrical capacity of the cable. Each device connected to the bus increase the electrical capacity on the wire, the length of the wire itself causes this. Increasing the capacity the wire will behave as a battery, needing longer transition time from an high value to low value.

4.1.1 Address Space

I^2C has two addressing schemas, 7 and 10 bits, giving space for 120 and 1016 addresses (excluding the reserved addresses). The I^2C extension uses exclusively the 7 bit format, and only to create groups or clusters of nodes. The addressing is based on a 16 bit identification number, called Client ID. How each client obtains it is explained in the ARP

Implementation section. Therefore each device (apart the System Host) is identified by a 16 bit + 7 bit (from I^2C) pair, making up a 23 bit addressing schema.

Not all the 16 bit client IDs are available. 64 are indeed used to implement a multicast address. Each client ID combination, whose 10 most significant bit are set to 1, are considered reserved. So the I^2C protocol stack will try to match any sequence like 1111 1111 11xx xxxx to its own multicast address, while the six least significant bits define the multicast group. Each client has to have a stack of 6 bit group id. Indeed each Client can register itself to more than one multicast group. 00 0000 is reserved and used as default value (no group registered). A Client are registered into a multicast group by the System Host. A multicast transaction is unidirectional, from the System Host to the Clients. Multicast addresses are issued after an I^2C general call, therefore any client grouped into any cluster (the 7 bit I^2C address), can receive it. No client can respond to a multicast, since the answer would have no meaning to the issuer (more than one slave could try to write on the bus at the same time). So the application protocol commands built upon a multicast should be designed as write only. However the Clients registered to a multicast group, can react to a multicast command, trying to start a transaction with the System Host.

Registration to a multicast group is not compulsory: should be used only when necessary, due to the high amount of traffic one multicast call can generate on the bus.

Why clustering? Clustering is meant as a way to save power and make the whole system more energy efficient. In some micro-controller the I^2C hardware is independent by the MCU core. This allows to put the core in deep sleep, while having the I^2C controller monitors the bus traffic. The I^2C Extension does not make direct use of the standard addressing schema. Therefore the only way to send an addressing command, is to use the general call. Anytime a general call is issued the controller raises an interrupt to wake up the MCU core to process the content of the packets. These might not be relevant to some slaves, but still all the slaves on the bus had to be waken up.

The I^{2}C bus extender heavily relies on the general call, to supply all its command and also to address clients. So any time the System Host, or some other client tries to address a device, all should wake up. Clustering allows the protocol to wake up only a subset of the clients on the bus. Such behavior is made possible using the standard 7 bit schema, which becomes part of the Client's ID. This subdivision is performed by the System Host, which takes care to have balanced clusters.

Also clustering allows to increase the chances the arbitration process goes to an end in a shorter period of time. Arbitration happens only when two masters are trying to access the bus: allows a master to win the bus and complete the transmission. From the I^{2}C specification [10] "This procedure (arbitration) relies on the wired-AND connection of all I2C interfaces to the I2C-bus". Arbitration is determined bit by bit: for each one wrote on the bus, the master checks if it matches what is on the bus. If there is no match, then that master loses the arbitration and steps back, turning into a slave device. Clearly if addressing a device had to happen any time by general call, the first byte written is the same for each master. Clustering increases the chances two masters are writing different data, making one of the two (or more) winning the arbitration earlier.

The I^{2}C extension introduces three new concepts about addressing:

- Cluster ID - is a group of nodes made using the I^{2}C 7 bit address. Is set to 1111 111 at start up, and reset by the System Host, to create balanced groups.
- Client ID - is the actual host identifier. Is a 16 bit value. When a Client matches its own Cluster ID with what issued on the bus, it will wait for the transaction master to issue a Client ID. After the match, the transaction continues with any

other protocol command.

- Multicast address - is a 6 bit long ID, in the form of 1111 1111 11xx xxxx. Is always issued after an I^{2}C general call (Cluster ID 0000 000). The clients will match the most significant 10 bits with a sequence of 1s, indicating that what follows is the Multicast ID. If there is a match the client will keep reading the bus.

The next section describes how a client gets assigned its own Client ID and Cluster ID. This also describes how the clients interact with the System Host.

4.1.2 Client and Cluster ID acquisition

To begin with the I^{2}C Extension does not make use of the Read direction. All commands are issued as Write. This is due to a technical restriction of the Read in the standard I^{2}C , where the transmission direction switches from 'sending master receiving slave' to 'sending slave receiving master' after the first data byte [10]. The I^{2}C Extension needs the first two data bytes to issue the Client ID, making useless the I^{2}C read. Therefore the protocol is asymmetric, and requests and answers happen in separate transactions.

As seen in the previous section each client is assigned a unique 16 bit ID. Alongside this assigned the normal 7 bit I^{2}C address, which will work as a Cluster ID.
The Client ID is self assigned by the Client (now called Address Requester) itself, and confirmed by the System Host. Client ID acquisition is a procedure that spans on two stages.

- The Client generates its own ID randomly.
- The Client ask the System Host to confirm its own Client ID. This to ensure is unique.

The first stage, the generation of the ID, can happen with any reliable random number generation approach. There are several options, such as the von Neumann's Monte Carlo

method, which can be combined with other hardware based generation. More about the topic will be coved in chapter ”Prototype Implementation”.
The second stage is more delicate, and its success is necessary to ensure unique Client IDs across the extended I^{2}C network. This process will introduce the use of states. The protocol is designed to act as a state machine. So each client will have one state per time, in which is able to perform some kind of operations, in this case Client ID confirmation. States are also necessary to keep track of replies to a command, due to the asymmetry of the I^{2}C Extender protocol.

It issues the Acknowledge ID command by: START condition, followed by the I^{2}C address of the system host (the standard address is 0001 111) and the write bit (0), then the bus command ’Acknowledge ID’ coded into 0x41 or 0100 0001, followed by the two bytes containing the 16 bit Client ID. If all the bytes are acknowledged, the Client enters the Acquisition state and sets itself the temporary Cluster ID 0001 110. Note that the System Host might decide to not acknowledge the bytes after the ’Acknowledge ID’ command, if it is already dealing with another Address acquisition process. In this case the not acknowledge Client has to step back and try later (10 seconds).

Table 4.1: Acknowledge ID

START	SYS HOST + W	A	0x41	A	Rnd Cluster ID
A	Client ID H	A	Client ID L	A	STOP

Table 4.2: Ping ID - Request

S	GC + W	A	0xC1	A	Client ID H	A	Client ID L	A	P

After this command, is sent, the Client ignores any packet sent to its ID, until it receives either a Regenerate ID or Valid ID command. These two commands are System Host only. In the meantime the System Host tries to discover on the network other devices

Table 4.3: Ping ID - Reply

S	SYS HOST + W	A	0xC2	A	Client ID H	A	Client ID L	A	P

Table 4.4: Regenerate ID, note that 0001 110 is the temporary Cluster ID.

START	0001 110 + W	A	0x44	A	New Cluster ID
A	New Client ID H	A	New Client ID L	A	STOP

with the Client ID it has just received by pinging that Client ID on the I^{2}C bus. If it gets any reply it means that ID was already taken by some other slave. If this happens the System Host issues a 'Regenerate ID' command followed by the Client ID. This command will be ignored by any Client that is not in the 'Acquisition' state.

The Ping command is made of two parts: a System Host ping request (see table 4.2), coded into 0xC1 or 1100 0001, issued by Cluster ID or GC. This is up to the state of the I^{2}C Extender protocol. In this case (Client ID acquisition and acknowledge) the ping request is sent by General Call. If any client should match the Client ID pinged with its own, then it answers by ping reply. Ping reply (see table 4.3) is the second part of the Ping command and is coded into 0xC2 or 1100 0010. It should do so by 500 ms, and all other host step back any Start condition for that period of time (after they received any ping request). The ping reply is sent at the System Host I^{2}C address, which is 0001 111.

If the System Host ping timer expires (after 500 ms), then it assumes nobody on the I^{2}C bus has got that Client ID. Therefore it consider the Client ID valid, and it confirm so at the client itself, issuing the command Valid ID coded into 0x43 or 1100 0011 (see table A.2). Valid ID is sent at the client using the pair temporary Cluster ID + Client ID + new Cluster ID. The new Cluster ID is part of the balanced groups discussed at the Chapter's Introduction. All the bytes have to be acknowledge to consider

Table 4.5: Valid ID, note that 0001 110 is the temporary Cluster ID.

START	0001 110 + W	A	0x43	A	New Cluster ID
A	Client ID H	A	Client ID L	A	STOP

the Valid ID command transmission as successful. If the transmission is not successful, then the System Host will try two more times, after which the Client ID + Cluster ID are discarded. If everything goes right the System Host will push the pair into a Clients stack.

In case the System Host ping does not expire and a reply is received, it means some other client has that Client ID. Therefore that Client ID should not be considered available, and the Address Requester should generate a new one. The System Host uses the command 'Regenerate ID' to instruct the Address Requester that its ID is already taken. This command is coded as 0x44 or 1100 0100 (see table 4.4) and is sent at the temporary Cluster ID + Client ID pair. At this point the Address Requester client should exit the 'Acquisition' state, and repeat the procedure from the first step, the random ID generation.

4.2 Multicast

Multicast is a new feature introduced by the I^{2}C Extension. It allows the System Host to create virtual groups of hosts, and to issue commands to them with one single transaction. Is composed of two commands: 'Set Multicast' and 'Unset Multicast'. These are issued to a specific client. Each Client can be registered in more than one multicast group, the limit is the memory of the client device. Each client device keeps a stack of multicast ids it is registered on. Each multicast id is 6 bit long. Setting a multicast address is a quite simple operation. The System Host writes on the bus, the Cluster ID, Set Multicast, Client ID, and the multicast group address. This command is coded as 0x45 or 0100 0101 (see table 4.6). Of the multicast group address only the least significant 6 bit are retained, and the

2 most significant are set 0 by default.

Table 4.6: Set Multicast group address

S	Cluster ID + W	A	0x45	A	Client ID H + L	A	00xx xxxx	A	P

Table 4.7: Unset Multicast group address

S	Cluster ID + W	A	0x47	A	Client ID H + L	A	00xx xxxx	A	P

To unset a multicast group address from a client, the System Host has to write the Unset Multicast command on the bus. Is also quite simple, the sequence is Cluster ID + Unset Multicast + Client ID + Multicast group address as in the Set Multicast command. The only difference is the command Unset Multicast, coded as 0x47 or 0100 0111 (see table 4.7).

Table 4.8: Write a command to a Multicast group address

S	GC + W	A	0x48	A	1111 1111 11xx xxxx	A	DATA	A	P

Writing to a multicast group is also very easy. Is a normal I^{2}C Extender write on bus command, the only difference is the Client ID, which uses the special Multicast ID code, which is 1111 1111 11xx xxxx the x' are replaced by the Multicast Group address the System Host wants to write to. The group 00 0000 is reserved and is indeed the standard group every host is registered to by default (it actually means no Multicast ID). There is no Cluster ID and the Multicast write has to be issued with a General Call. The command is coded as 0x48 or 0100 1000, In table 4.8 is shown Multicast Write.

4.3 Bus channel multiplexing

The I^{2}C bus channel multiplexing is a feature which directly involves only the System Host. Is also the only feature to require extra hardware other then the I^{2}C hardware controller. It is based on an IC which allows to multiplex up to four separate bus, to support mixed voltage levels, communication speed and to resolve address conflicts on standard I^{2}C devices. It is also very useful to increase the number of devices available, due to the full separation of the four busses (from here channels). These channels behaves as independent busses and are set active only one per time by a bus master. However for some applications, such as those mentioned in the earlier chapters, is necessary to have a more flexible allocation procedure of the channels. The proposed I^{2}C Extension protocol aims to transparently allocates the System Host time to each channel, in a completely transparent fashion. The four or more channels, are therefore treated as a single virtual bus. In Section 4.4.1 is explained how each channel is scheduled to the System Host.

The most common I^{2}C multiplexer is the PCA9544, the I^{2}C Extender protocol is implemented on this integrated circuit. This IC has one upstream I^{2}C interface which fans out to 4 downstream SDA/SCL pairs also called "channels". The System Host is connected to the upstream interface. The channels are 4 independent I^{2}C buses which can be selected as active by the "master" device, in this case the System Host. The channel selection is done by writing a byte to the control register of the multiplexer. Such control register is as described in Table 4.9.

Table 4.9: The PCA9544 control register.

7	6	5	4	3	2	1	0
INT3	INT2	INT1	INT0	X	B2	B1	B0

Bits 7 to 4 are read-only interrupt flags, these are set to 1 when an interrupt is raised by a device on the respective I^{2}C channel. This feature is not used by the I^{2}C Extender. Although using the interrupt handler would have simplified the protocol design, it needs some extra hardware which is contrary to the design requirements.

Bit 3 is not used. Bit B2 when set to 1 enables the channel set in bits B1 and B0.

The channel determined in bits B1 and B0 is selected only when a STOP condition is placed on the upstream bus, to ensure all the SDA and SCL lines are in HIGH state [13]. This precaution should avoid that any false condition is written on the bus.

The next section introduces the Channel Scheduling protocol, and how it avoids that any false condition is generated on the downstream interfaces of the multiplexer. This is not an unlikely event, since the whole system is multi-master from an I^2C point of view.

4.4 Bus Multiplexing Commands

The I^2C Extender aims to support I^2C bus multiplexer in transparent way, to join all the channels into a single virtual bus. To support this feature the System Host is scheduled to each channel in Round Robin, allowing 250 ms to each channel, before switching to the next one. To select a channel the System Host write a configuration byte on the control register of the PCA9544. To do so it issues the sequence of Table 4.10;

Table 4.10: Set the PCA9544's Control Register to activate channel 1.

Start	Address + W	Ack	EN bit + Ch	Ack	Stop
START	1110XXX + W	A	0000 0101	A	STOP

The channel becomes available after the Stop condition. The System Host has to notify all the Clients connected on that channel that they can communicate with the central node. To do so the System Host issues, in general call, the channel active message coded as 0xAA or 1010 1010, see Table 4.11. Once the Clients receive this message they can communicate with the System Host. A window of 250 mS is available before it shifts to the next channel. This command is received only by the Clients of the currently active channel.

Once the time frame scheduled for that channel has expired the System Host will notify all the Clients writing the deactivation message on the bus, see Table 4.12 for the

Table 4.11: System Host notifies all Clients that their channel is now active by sending the following bytes.

Start	General Call	Ack	Enable Command	Ack	Stop
START	0000 0000	A	1010 1010	A	STOP

message structure. The deactivation command is coded as 0x55 or 0101 0101.

Table 4.12: System Host notifies all Clients that their time frame is finished, and that the next channel is being activated.

Start	General Call	Ack	Disable Command	Ack	Stop
START	0000 0000	A	0101 0101	A	STOP

The time frame can be extended to complete an ongoing transaction. If the timer timeouts, and a transaction is not completed (e.g. has been received a command, but a reply is not yet issued), the System Host pushes the pending message into a stack, to be executed once the relative channel is scheduled again. When the System Host schedule a channel for which there are pending transactions it will resume them before to send the "Channel Enabled" message. This is essential to ensure the System Host has absolute precedence on bus acquisition, and to avoid the Clients to resend the request due to timeout of the previous. Such policy will reduce the overall traffic on the bus.

This is not a fair scheduler. In Section 6.2 are described few improvements to increase the fairness of the system. The selection of the channel, happens through the standard I^2C transaction, as seen in the previous section.

4.4.1 Bus Channel Scheduling

The channels are scheduled by the System Host in a Round Robin fashion. Each channel is set active for a certain time frame, decided by the System Host. This timer is started as soon the System Host sends the "Channel Enabled" message. When this timer times out, the System Host will issue the command "Channel Disabled" and will schedule the next channel regardless of any pending transaction. If any request has not been satisfied, this will be pushed into a stack and resumed the next time that channel is selected. Such a scheduling policy is not fair. This is also due to the bus acquisition process the Clients have to go through. A Client might loose the bus arbitration several times, before it can contact the System Host. This becomes even more likely when the bytes it has to transfer contains many zeros. It might also happen that this Client get to complete a transaction only few instant before the timeout of the time frame. The System Host will move on, scheduling the next channel, and not satisfying that request. Also other clients on the other channels can be eager to contact the System Host as soon possible, and before a transaction is completed successfully can pass an undetermined time, leading to data starvation. Such a scenario becomes more likely in over populated busses. The I^{2}C Extender tries to reduce the likelihood of this situation by completing all the suspended transactions before to notify the Clients their channel is active. This is not a full solution but is compatible with the requirement of simplicity of implementation, in section 6.2 are discussed better solutions.

4.4.2 Commands and States

In the past sections were introduced the commands of the I^{2}C Extender. However the commands itself are only a part of the I^{2}C Extender. To each command, either received or sent, is associated a state. Indeed the whole system is a finite state machine. Such an approach is also used by the vast majority of the I^{2}C hardware implementation [2, 9], revealing itself as the easiest way to keep different systems (the Clients and the System Host) consistent to each other. Whenever a Client or the System Host send a command, they enter a particular state, either to receive a reply or to accept incoming commands. In the next chapter will be covered in the detail the state machines associated with each command.

Chapter 5

Prototype Implementation

This chapter is meant as a documentation of the prototype design. It shows what the various components of the software are for and how they interact to each other. The I^{2}C extender works at a very low level, therefore each section will cover the interaction of hardware and software. For example will be explained how a provisory client ID is generated randomly using an ADC converter, or how the System Host allocates CPU time for the multiplexer channels. Said description is split in two separated sections, one for the software running on the clients and the other covering the System Host software.

A similar structure is kept for the hardware sections, where is introduced the hardware designed for the prototype. The section about the client design is more vast than the one about the System Host. The client hardware has been designed from scratch, while the System Host is a standard ARM7 development board. Still some custom System Host hardware is present, mainly regarding the wiring of the I^{2}C multiplexer.

Section 5.5, will provide an overview of I^{2}C used over long wires ($\gg$10 Meters). Most of the section will provide considerations about the I^{2}C Extender implemented on a bus setup as described in the application note about long distance communication [8].

5.1 Software Design - Client

The next subsections go through the most important I^{2}C Extender procedures performed by the Client ID. These explain how the Protocol introduced in Chapter 4 have to be implemented. Where strictly necessary is made use of finite state machines.

5.1.1 Acquire ID

The most critical phase the I^{2}C Extender has to cope with is the Client ID and Cluster ID acquisition. In the worst case scenario as soon the bus is powered up all the clients try to access the the bus at the same time. The standard I^{2}C mechanisms, bus arbitration and clock synchronization, are not sufficient. One of the priority is to make the arbitration happen as soon as possible. Therefore increasing the entropy writing some random data on the bus might be of help. Before to try any bus acquisition every Client generates 3 random bytes. One is stored in the I^{2}C address register of the micro-controller, and is the random Cluster ID, the other two bytes are used as Client ID High and Low. So there are 24 random bits, leading to 16.777.216 possible combinations. In Table 4.1 is clear that these bytes are sent relatively early, after the Acknowledge ID command (0x41). This helps the arbitration.

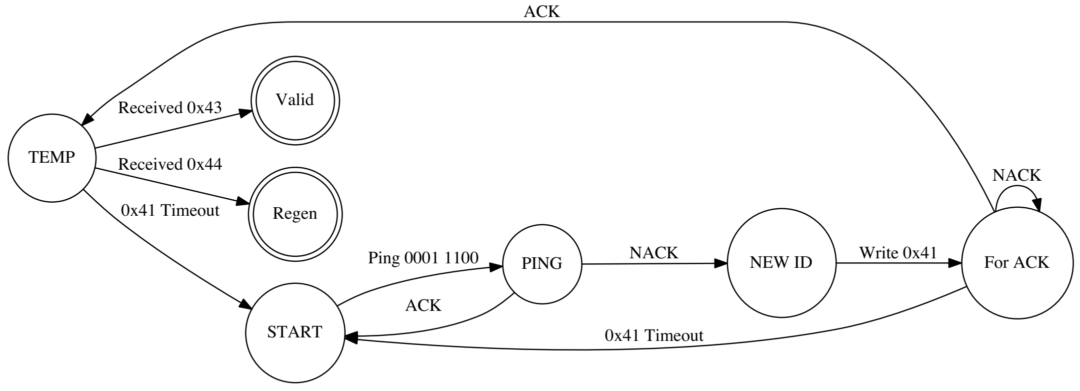


Figure 5.1: The Client's Address Request procedure finite state machine.

However this happens only after the Client has made sure the Temporary Cluster ID is not taken and available. The Cluster ID is reserved I^{2}C address (0001 1100) whose scope is to create a clear "area", where to hold the communication with the System Host wile waiting for it to assign it an available Client ID. The Temporary Cluster ID is reserved for only one Client per time. Before to send the 0x41 command, the Client interested pings the Temp Cluster ID with a standard I^{2}C transaction. If it gets no answer (NACK) then it issues a repeated START condition (do not release the bus), and sends the 0x41 command, writing also the random Cluster ID and the two random Client IDs. At this point the arbitration should have happened. The transaction continues if all the bytes of the 0x41 command are acknowledged. If not, the state machine is reset to state START. Otherwise it goes in TEMP state, where it will set immediately its Cluster ID to 0001 1100 and waits for an reply from the System Host. If the transaction timer (500 mS) should timeout, the state machine is reset to START. It carries on to accepted states only if it receives the message 0x43 (Valid ID) or 0x44 (Regenerate ID). Whichever it receives it has to set Client ID and Cluster ID to the new values. This double state is intended to keep track of changes made at the Client ID by the System Host. It is more a debugging feature than something necessary.

5.1.2 Ping

The Ping command (coded as 0xC1) is a powerful management feature. This Ping command pings the 16-bit Client IDs. To ping the standard 7-bit addresses (such as the System Host or the Temporary Cluster ID) the I^{2}C command must be used. It is mostly used by the System Host to know the status of a Client. If this does not reply to any Ping, it is probably off-line therefore might be removed from the System Host's Clients list. A state machine is involved only when transmitting a ping request. In the other direction there is no need of a state machine. The interested Client simply sends back the command 0xC2. In case a Client needs to know about the state of another Client it has to write the ping 0xC1 command (and win the arbitration), then it waits for a reply. It must allow 500 mS for the other party to reply. The fact a the other part has acknowledged all the bytes transmitted is already a good point, but to be sure the request was actually processed is better to wait to receive the message 0xC2. If no reply is received after 500 mS then the ping is KO.

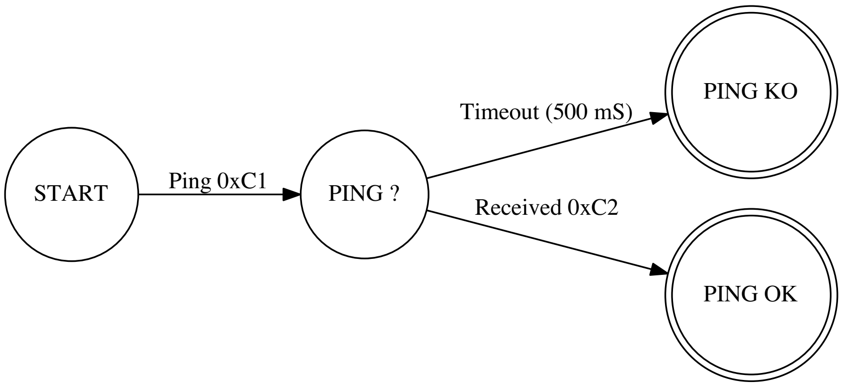


Figure 5.2: The Client's Ping Request procedure finite state machine.

5.1.3 Multicast

The multicast procedures do not base their mechanism on any internal finite state machine. The communication is direct and considered failed if not acknowledged. When the System Host addresses a Client to register it into a multicast group it gets a response immediately, so it does not have to keep track of this transaction's state while dealing with other transactions. The Set Multicast Group command (coded as 0x45) is considered transmitted and executed successfully whenever all the transmitted bytes are acknowledged. The client on its side receives two bytes of Group ID of which stores the second byte (only the 6 least significant bits are relevant). When the client receives, via General Call, the command 0x48, it will and mask the second byte with the stored byte. If there is a match, then the client will process the following bytes, otherwise it ignores them. The Unset Multicast Group command (coded as 0x47) works pretty much the same way as the Set Multicast Group. The transaction is considered successful only if each byte is acknowledge. When a Client receives this command it keeps the second byte of the multicast address, seeks it in its list of multicast groups. If any it deletes the entry. If no group is found then it ignores the request. Clearly such approach is not failure proof: the System Host has no certainty that either the "Set Multicast Group" or the "Unset Multicast Group" once received are actually processed. However, due to the constrained bandwidth of the I^2C bus, it is worst to flood the bus with control packets, so this uncertainty is the least worst problem.

5.2 Hardware Design - Client

The Client hardware passed through several different stages. The first steps of the Client software development was made on the common Arduino Duemilanove AVR development platform. It is programmed in plain C. Contrary to what originally planned no operating system is used. The Arduino platform is very convenient to prototype, it offers a micro-controller with all the extra hardware needed to program and power it up. It has an LED on one of the output lines, and a power regulator circuit, it can be programmed through the serial port. The low price is another good reason to choose it. The micro-controller installed is an ATmega328 running at 16MHz through an external oscillator, it has 2 KBytes of ram, 1 KByte of EEPROM and 32 KBytes of in-circuit programmable flash memory. The board comes with a very handy ICSP (In-Circuit Serial Programming). Through this 6 pin interface is possible to flash new software on the micro-controller. It is also possible to debug the program, this family of MCU supports the debugWIRE protocol, to support program flow control using one single wire and one pin of the micro-controller (specifically the RESET pin). To use this feature is necessary to have an hardware programmer that supports it, such as the AVR Dragon. To enable the various features the ATmega328 supports it necessary to "burn" the correct fuses, which are flags of a configuration register. This is done through the ISP interface. This operation is delicate, because if mistakenly the ISP is disabled, it is not be possible to re-enable it unless through high voltage programming [2]. The software development was delayed due to errors programming these fuses.

The first phase was to use the Arduino Duemilanove as a way to learn the ATmega328 programming. The second step was to create a custom hardware where to run the I^{2}C Extender client software. As stressed in Chapter 1 one of the requisite is to run the software on commercially available hardware and possibly very simple and low cost. The final design, is indeed very simple and is made of the bare minimum components needed to make the ATmega328 run. From the schema, in Figure 5.3, is possible to appreciate the low complexity of the hardware needed to run the I^{2}C Extender.

The most notable parts are few decoupling capacitors, C1 is needed to stabilize the current flow that powers the system. While capacitors C2 and C3 are necessary to separate the crystal Q2 from the ground, and make it less sensible to sudden variation of current. The crystal is a normal tin encapsulated component which provides a clock of 16 MHz. On the top part of the schema is possible to see the ISCP connector, to program the flash and EEPROM memory. As mentioned earlier is possible to debug the software from this interface. On the bottom right side of the schema there are a yellow and a green LED, originally intended to signal the status of the client. Currently the software does not make any use of them. Above the LEDs there are 4 pins. The two labelled SDA and SCL go to the I^{2}C bus. No pull up resistors are needed, these should be installed nearby the PCA9544, the most logical location. If each client carried its own pull up resistors the two I^{2}C line would always have a voltage too low to be recognized as HIGH.

The schema in Figure 5.3 was derived from the Arduino Duemilanove itself, stripping away all the non necessary components. The ATmega328 data-sheet [2] gave further hints about what is strictly necessary to run this micro-controller. These parts were the USB to RS232 converter and the circuitry to program the flash through the boot-loader (each Arduino has a boot-loader, which is absent on these clients). Also the power regulator was unnecessary, at least for the scope of the prototype, since all the devices are the same

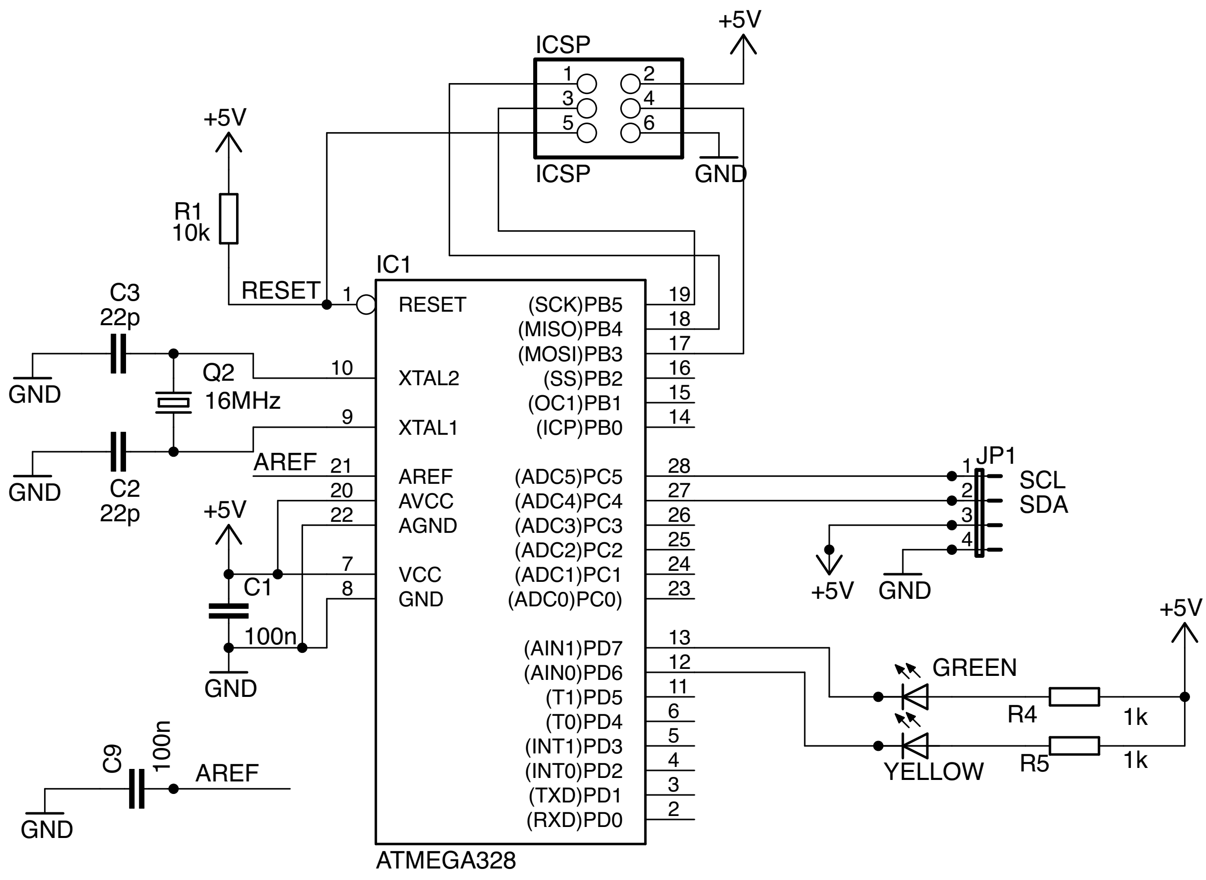


Figure 5.3: The client schema is very minimal, there are just the bare minimum components to run the ATmega328. This simple design has been chosen to stress the idea that the I^{2}C Extender is intended for extremely low power and low cost hardware.

and they take power from a common VCC line. The VCC line is stable at 5 Volts by design.

However this prototype did not function as expected. This is due to the not very precise assembly of the prototypes. After several attempts to make it working the best option seemed to revert to the standard Arduino. Lack of time and experience on hardware design and assembly were key to make this decision. Nevertheless the proposed hardware design should fit the purpose here described.

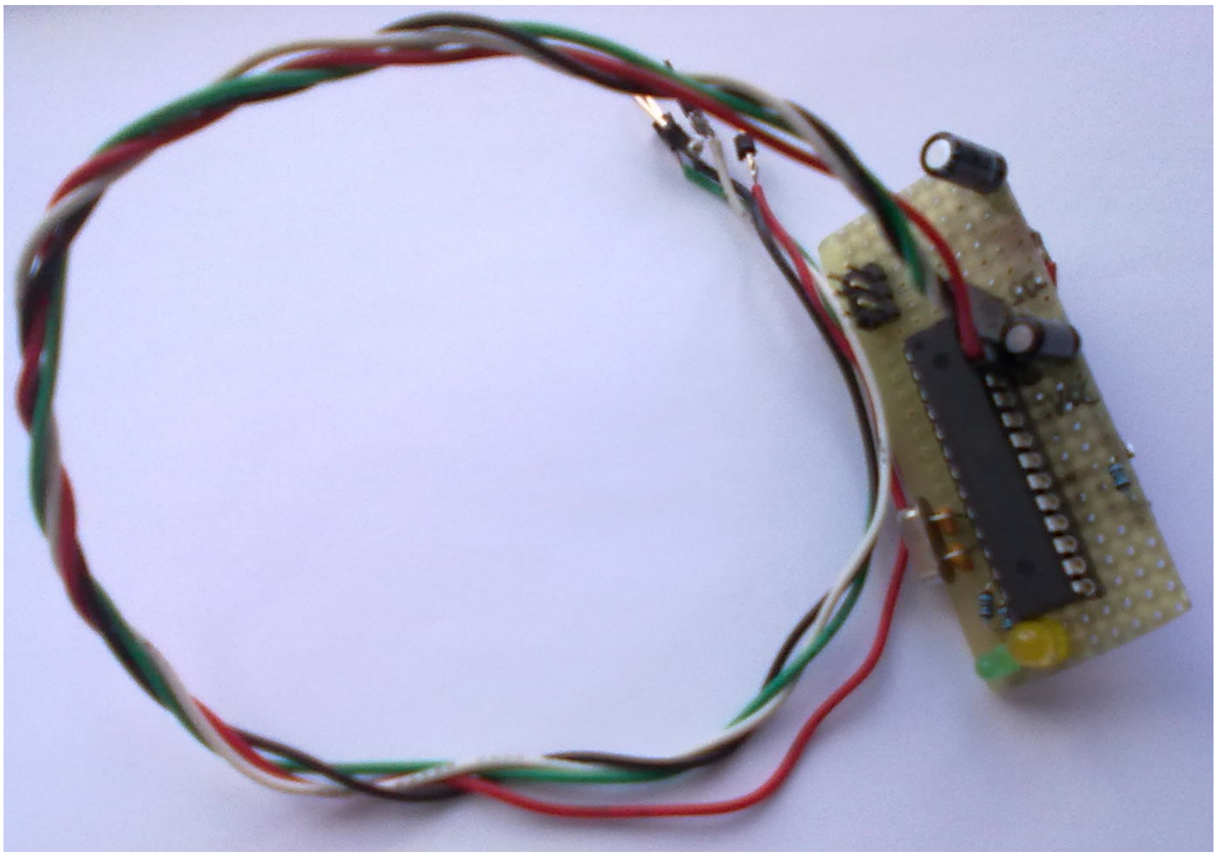


Figure 5.4: The Figure shows a photo of one of the earlier phisical prototypes of the custom hardware client. This simple attempt to build a custom hardware from scratch did not work as planned.

5.3 Software Design - System Host

This section is the equivalent of section 5.1 for the System Host. The ping command is not covered, since what is valid for the Client is valid for the System Host. However the subsection 5.3.1 goes through the System Host side mechanism of Client ID assignment.The System Host has been implemented as a FreeRTOS task which shift channels whenever a time interrupt is raised. This task also reacts at the interrupts coming from the I2C0 controller of the LPC2468 board. The software prototype can assign Client IDs, but makes no use of the Multicast feature. The prototype is very simple, but can use the I^{2}C Extender features without problems.

5.3.1 Address Assignment

Once the System Host has issued the "Channel Enabled" message it waits for incoming messages or commands. Very likely the most common message is "Acknowledge ID" coded as 0x41. On receiving it processes it by selecting the two bytes composing the Client ID (most significant and least significant bytes). If this Client ID is not already assigned or reserved (a stack of IDs is kept by the System Host) it will go on by allocating this Client ID into a Cluster ID, being careful this is not overcrowded. Each Cluster ID should be well balanced. A this stage the System Host prepares a message "Valid ID" 0x43 and sends it back to Client. If the Client ID was already taken or part of a set of reserved Client ID (such as a multicast group) it generates ex-novo a Client ID and Cluster ID pair, then it sends back a "Regenerate ID" 0x44 message.

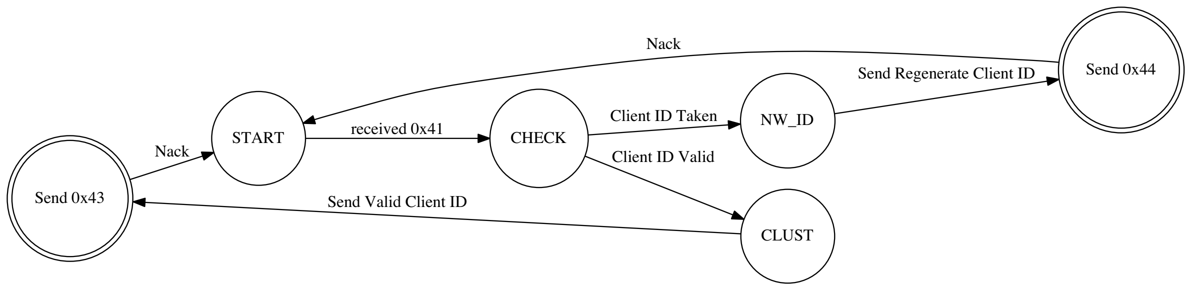


Figure 5.5: This finite state machine diagram shows the Client ID assignment from the System Host's point of view.

If any byte is not acknowledge the transaction is interrupted, and the System Host sets ready to accept new incoming requests. Should this happen it is duty of the Client to re-issue the command "Acknowledge ID" 0x41. The assignment procedure is fairly simple, also due to the fact that the System Host has no knowledge about the nodes on the network, and the fact it is not the initiator of the transaction (so it does not have to go through the bus arbitration).

5.3.2 Multicast Groups

Also here the Multicast management is very simple. There is no certainty that a Client actually processes a Multicast Set Group or Unset Group command. The only safe procedure the System Host has is to reissue a command if any of its Bytes is not acknowledge. The other commands do not expect any reply, due to the reasons explained in Section 5.1.3.

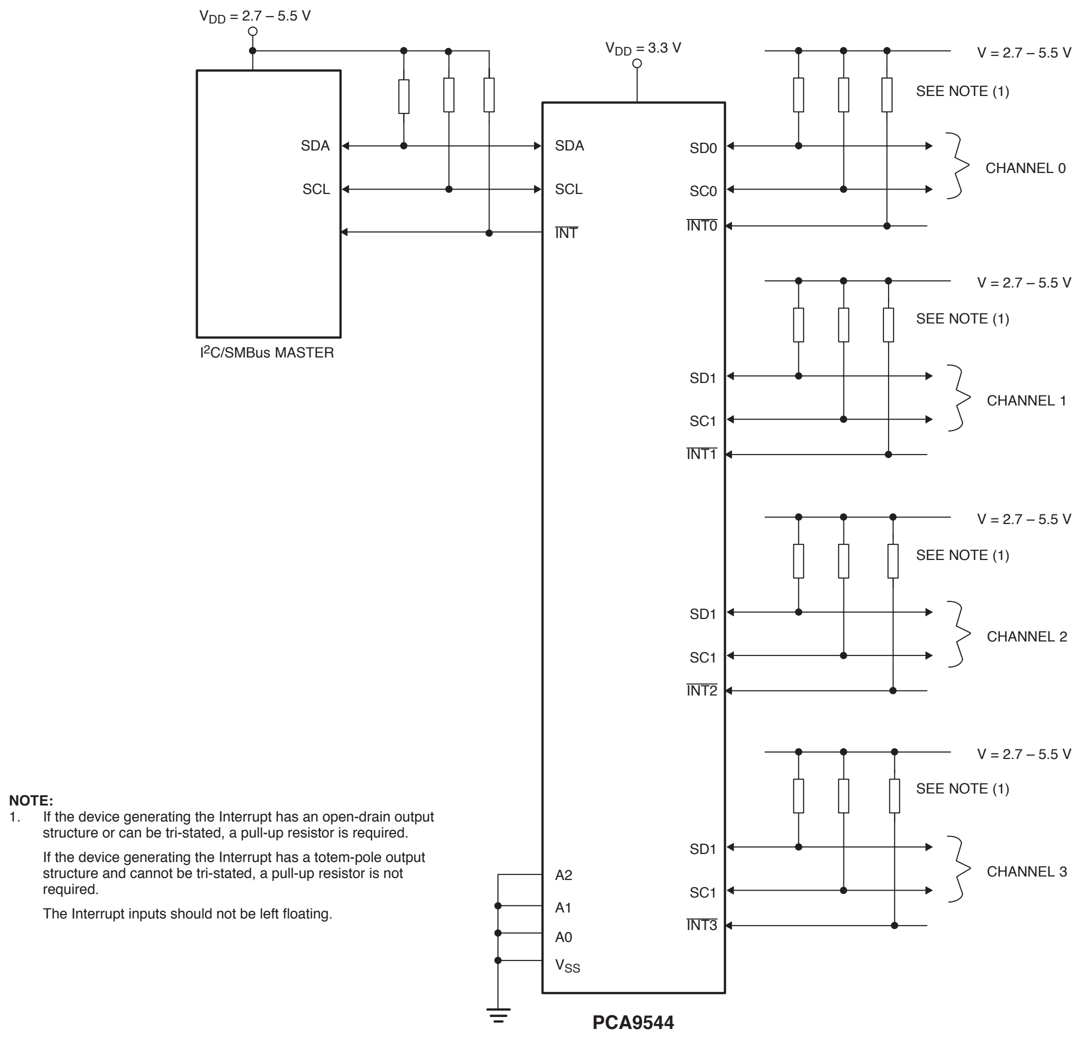
5.4 Hardware Design - System Host

The System Host hardware was a standard LPC2468 board. It has three I^{2}C controllers, one of them supports the whole I^{2}C specification [9]. This controller was chosen to play the role of System Host. From an hardware point of view it means that two trailing wires have had to be soldered to pin P0.27 (SDA 0) and pin P0.28 (SCL 0). These two trailing wire are connected to a the SDA and SCL pins of the PCA9544, also are connected to VCC with pull-up resistors.

Figure 5.4 helps to understand how to wire the PCA9544 I^{2}C multiplexer. The master in the diagram is the I^{2}C Extender System Host, which enables the downstream channels, where the I^{2}C Extender Clients are connected. The interrupt lines are not used to keep the I^{2}C Extender as close as possible to the requirement about standard hardware. However in Section 6.2 is proposed an approach which makes use of the Interrupt lines to improve the fairness of the channel scheduler, and therefore avoid data starvation.

5.5 Long Distance Communication

One of the most challenging requirements of the I^{2}C Extender is to support its function also over long wires, over 10 Metes. As mentioned in Chapter 1 the main problem of long wires is the fact they tend to increase the electrical capacitance of the system. The capacitance is also increased by each device connected to the bus. The overall amount should not be grater than 400μF, practically limiting the length of the wire to few meters. However there some solutions to this problem. All of these are pure hardware approaches, so they go beyond the scope of this dissertation, which is mostly focused on software. The first approach is to use an I^{2}C buffer such as the P82B715. This is a bipolar integrated circuit which reduce the electrical capacitance by a factor of 10 and allows the whole system to be around 3000μF [11]. This is the simplest solution. The second solution is much more complex and makes use of twisted pair of wires for the SCL and SDA lines. To this wires are connected two P82B96 buffers, one for each end. This approach ensures high speed, and a good link quality over very long distances (also over 100 meters [8]), but it limit the number of nodes to only two. So this second solution is not good for the I^{2}C Extender, the first is the only viable. However having abandoned the option of custom hardware, as mentioned in Section 5.2, the P82B715 buffer was not tested.



NOTE:

1. If the device generating the Interrupt has an open-drain output structure or can be tri-stated, a pull-up resistor is required.

   If the device generating the Interrupt has a totem-pole output structure and cannot be tri-stated, a pull-up resistor is not required.

   The Interrupt inputs should not be left floating.

Figure 5.6: The Figure shows the PCA9544 wiring diagram. This makes clear how this I^{2}C multiplexer plays the role of hardware connector among the System Host and the Clients. In the System Host is the Master, while the Clients should be connected to the downstream I^{2}C channels. This schema is quoted from the PCA9544 technical data-sheet [13].

Chapter 6

Results and Conclusion

In this final chapter are commented the achievements of the dissertation. There are notes about the issues encountered during the development of the prototype, either hardware and software. In addition each I^{2}C Extender requirement, presented in Chapter 1 is evaluated against the results obtained by this dissertation project. For each point that did not succeed as planned is explained what went wrong, or why such part or requirement became irrelevant. This final analysis goes towards the last section of this Chapter, Section 6.2, where are proposed some improvements to the I^{2}C Extender.

6.1 Prototype Efficiency

6.1.1 Dynamic Addressing

This is most important and challenging feature of the I^{2}C Extender. What made it more complex was the fact that no ARP techniques could be used. To do ARP the Client device should have a fixed address, to contact it at. This fixed address makes it unique on the bus, and simplify the process of bus arbitration and synchronization. This had to be worked around using a random generated sequence of bytes. Specifically are random generated the Cluster ID and the two bytes of the Client ID. The random numbers are generated reading from an analog digital converter and using that number as seed for a semi random number generator. It takes 24 reads to complete the whole three bytes. The new use of the old I^{2}C 7-bit address as Cluster ID is also quite smart. The I^{2}C Extender relies on the General Call for several command or messages. The original design intended to use the General Call for each command, independently by the direction. This was necessary, to not use the old address space and create one completely new. However further thoughts gave the idea to use the existing address schema to make more efficient the new one. The original plan was to carry on the test with 10 Client devices, but unfortunately this was not possible. The test was carried on with only 1 Client, an Arduino Duemilanove. An attempt to use also an Arduino Mini Pro did not do well, due probably to the different operating voltage (one runs at 5 Volts while the other at 3.3 Volts). The description of this portion of the protocol is clear and detailed enough to reproduce the protocol on different hardware or software.

6.1.2 Multicast

While the Cluster schema is fixed by the System Host, and is heavily connected to the I^{2}C Extender mechanisms, the multicast follows a different grouping approach. It is intended to work exactly as the Ethernet multicasting, where clients can register to a multicast group. So the I^{2}C Extender Multicast groups the Clients by logical differences, such as technical characteristics or sensing capacity. This feature can be used to synchronize a set of nodes, or to set a certain parameter equal to all them all. The multicast registration procedure is not fail safe, and although the command's bytes are all acknowledged the Client might fail without anyone noticing it. However this lack of flow control is the best trade off, in terms of bandwidth occupied and performance. The test of this feature were not extensive, mainly due to the lack of nodes to use as testbed.

6.1.3 Protocol Fairness

As already mentioned in other Chapters of the dissertation the I^{2}C Extender is not a fair protocol. This is mainly due to the fact that I^{2}C is not a fair protocol either. Is very hard to build a fair mechanism with such this constrained resources. A Client, particularly on an overcrowded bus, might starve for data. It is possible a Client keeps loosing the arbitration, making it impossible to gain access to the bus. This might happen when a Client tries to write a byte with a long sequence of zeros, it is very hard for it to win the arbitration. Also the multiplexer scheduler is not very fair. It does not implement any priority system, and the round robin scheduler, although might sound fair enough, does not help. This is due to the lack of priority queues, so a Client or group of Clients might have to wait also a very long period of time before get to initiate a transaction with the System Host. Sadly all these issues could not be experimented in detail, due to the small number of physical devices available.

6.1.4 Long Distance Communication

Stretching the limits of I^{2}C also in terms of communication range was the most intriguing challenge. It is about reducing the electrical capacitance of the wire to allow longer wires or more devices connected. This is a fully physical issue. There is no way to intervene on it via software. Some possible approaches are analyzed, so it is possible to integrate the I^{2}C Extender with some hardware buffer.

6.2 Future Works and Improvements

At the light of the previous considerations is possible to think at several future developments for the I^{2}C Extender.

6.2.1 Channel Proxy

A proxy could help to increase the fairness of the channel scheduler. Such an improvement needs some extra hardware, but the benefits might be very important. The idea is to have a fixed device (pretty much like the one proposed in 5.3) connected directly to the downstream I^{2}C busses of the PCA9544. This fixed client is also connected to the interrupt lines of the I^{2}C multiplexer. This device is alway active, also when the System Host is busy serving other channels. The Clients in need to communicate with the System Host will not try to contact it directly, as it is done currently, but will send the message at the Channel Proxy. The Channel Proxy queues up all the messages, in order of arrival. As soon it receives a message it triggers the interrupt line associated with its Channel. So the System Host knows that in that channel is happening something it should be aware of. The Channel Proxy will take care of forwarding all the requests at the System Host. This would turn the channel scheduler from Round Robin to interrupt based. Such a scheduler might be good also for real time applications. Clearly the introduction of a Proxy steers the I^{2}C Extender a bit off the original requirement to not use any extra hardware. However for specific application, this system might be the difference among using this bus technology or try something else. The Channel Proxy can be the the solution to data starvation, it would make also a better scheduling policy being more reactive to the needs of the system.

6.3 Conclusion

Concluding the I^{2}C Extender is an nice elegant solution to an old problem of the I^{2}C users. The prototype here proposed needs more testing, possibly with an high number of nodes. However it proved to be functional, and serves the scope of auto-addressing devices quite well. Few other features such as the Multicast and the integration of the hardware I^{2}C multiplexer increase the flexibility of the systems. The I^{2}C Extender scales up well, with 16-bit address space there is enough room for even more nodes than what are electrically compatible.

The future work section gives a good introduction to the concept of Channel Proxy, which would improve the fairness of the system although built on a protocol traditionally not fair. This part is the most interesting to develop in future because it needs some extra protocol, and the challenge is to keep it as simple as possible as it is now.

Appendix A

I^{2}C Extender Commands

Table A.1: Command Codification

Command Name	Hexadecimal	Binary	Reference
Ping Request	0xC1	1100 0001	Table 4.2
Ping Reply	0xC2	1100 0010	Table 4.3
Acknowledge ID	0x41	0100 0001	Table 4.1
Regenerate ID	0x44	0100 0100	Table 4.4
Valid ID	0x43	1100 0011	Table A.2
Set Multicast	0x45	0100 0101	Table 4.6
Unset Multicast	0x47	0100 0111	Table 4.7
Write Multicast	0x48	0100 1000	Table 4.8
Channel Active	0xAA	1010 1010	Table 4.11
Channel Disabled	0x55	0101 0101	Table 4.12

Table A.2: Reserved Addresses

Host Name	Use	Address	Reference
System Host	Default System Host's address	0001 1110	Section 4.1.2
Temporary Cluster ID	A Client in Acquire ID state has its own Cluster ID set to this address	0001 1100	Section 4.1.2
Multicast ID	The 6 LSB explicit the multicast group	1111 1111 11xx xxxx	Section 4.2
No Multicast ID	This 2 byte ID is reserved and stand for no multicast	1111 1111 1100 0000	Section 4.2
General Call	General Call address to send a transmission to all nodes	0000 0000	See [10]
10-Bit addressing	To address a device using a 10 bit address, not used by I^2C Extender reserved by I^2C	1111 0xx	See [10]

Appendix A

Glossary

Short Term	**Explanation**
System Host	A permanent central node, whose role is to coordinate the I^2C Extender bus operations.
Client	Any I^2C Extender compliant device, being plugged into the bus. They are subordinated to the System Host commands.
Client ID	Is a 16 Bit long ID used to extend the standard I^2C 7 Bit addressing schema.
Cluster	These are groups of Clients populated by the System Host, which uses the old I^2C 7bit addressing schema to improve energy efficiency and reduce the network overhead.
Cluster ID	This is the standard I^2C 7 Bit address, used by the I^2C Extender in the clustering mechanism.

Bibliography

[1] Auke Jan ljspeert Alessandro Crespi. Amphibot ii: An amphibious snake robot that crawls and swims using a central pattern generator. In *Proceedings of the 9th International Conference on Climbing and Walking Robots*, School of Computer and Communication Sciences, Station 14, CH-1015, Lausanne, Switzerland, September 2006. Ecole Politechnique Federale de Lausanne (EPFL).

[2] Atmel Corporation. *ATmegATmega48A/48PA/88A/88PA/168A/168PA/328/328.* Atmel Corporation, 2325 Orchard Parkway, San Jose, CA 95131, USA, 8271c–avr–08/10 edition, 2010.

[3] Fujitsu Ltd Intel Corporation Linear Technology Duracell Inc, Energizer Power Systems Inc. *System Management Bus (SMBus) System Management Bus (SMBus) Specification.* SBS Implementers Forum, www.sbs-forum.org, 2.0 edition, 2000.

[4] N. Gautam, Won-Il Lee, and Jae-Young Pyun. Track-sector clustering for energy efficient routing in wireless sensor networks. In *Computer and Information Technology, 2009. CIT '09. Ninth IEEE International Conference on*, volume 2, pages 116 –121, 2009.

[5] Y. Kawahara, M. Minami, H. Morikawa, and T. Aoyama. Design and implementation of a sensor network node for ubiquitous computing environment. In *Vehicular Technology Conference, 2003. VTC 2003-Fall. 2003 IEEE 58th*, volume 5, pages 3005 – 3009 Vol.5, 2003.

[6] Akira Fuyuno Kei Okada et al. Device distributed approach to expandable robot system using intelligent device with device distributed approach to expandable robot system using intelligent device with super-microprocessor.

[7] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8 pp., april 2006.

[8] NPX, AN10658. *Sending I2C-bus signals via long communications cables*, rev. 01 edition, 02 2008.

[9] NXP. *LPC2468 User manual.* Philips Semiconductors, rev. 01 edition, December 2006.

[10] NXP, UM10204. *I2C-bus specification and user manual*, 2.1 rev. 03 edition, 06 2007.

[11] NXP. *P82B715 I2C-bus extender.* Philips Semiconductors, rev. 08 edition, Novembre 2009.

[12] Y. Ohmura, Y. Kuniyoshi, and A. Nagakubo. Conformable and scalable tactile sensor skin for curved surfaces. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 1348 –1353, May 2006.

[13] Philips Semiconductors. *PCA9544 4-channel I2C multiplexer with interrupt logic*, 06 2002.

[14] Maxim Integrated Products. *iButton^{TM} Overview.* Maxim Integrated Products, 081297 53/151 edition.

[15] Maxim Integrated Products. Guidelines for reliable long line 1-wire® networks, September 2008.

[16] X. Righetti and D. Thalmann. Proposition of a modular i2c-based wearable architecture. In *MELECON 2010 - 2010 15th IEEE Mediterranean Electrotechnical Conference*, pages 802 –805, 2010.