Technical **U**niversity of **C**rete

Electronics & Computer Engineering Department

Microprocessor & **H**ardware **L**aboratory

Diploma Thesis

Interface and wireless embedded applications of Bluetooth, based on microcontrollers.

by Christos Strydis

Supervising Professor: Professor Apostolos Dollas

Committee: Professor Apostolos Dollas
Professor Michalis Paterakis
Assoc. Professor Dionisios Pnevmatikatos

June 2003

«Εν τη υπομονή υμών
κτίσασθε τάς ψυχάς υμών...»

Dedicated to my parents…

Αφιερωμένο στους γονείς μου…

Contents

Acknowledgements

Getting involved with a newly-born technology -such as Bluetooth- is never easy. Relative documentation is limited and often erroneous while the accompanying hardware is seldom bug-free due to the lack of experience that only time can bring to designers. Enthusiasm and hard work are definitely key elements to a successful thesis but appear to be feeble friends when the various "irrational" development problems seem to pop in. In such cases, the advice and experience of other people is the best friend one can have.

For this reason and many more I would -first of all- like to thank my supervising professor Apostolos Dollas for his continuous support and insight throughout the development of my thesis and for his guidance throughout my academic years. Also, assoc. professor Dionisios Pnevmatikatos for his useful recommendations mainly during the first but decisive stages of development and professor Michalis Paterakis for making the proper arrangements with Ericsson to donate two "Application & Training Toolkits" to the M.H.L. long before I knew how to even spell 'Bluetooth'. Of course, many thanks should be directed to Ericsson Hellas and Ericsson for donating the two Kits and, thus, making the realization of this thesis possible.

I would also like to acknowledge Markos Kimionis, member of the technical stuff of the MHL, for providing me with all the required gear, technical advice and spirit throughout this thesis, and Ioannis Chatzakis, member of the teaching stuff of the Technical Educational Institute of Chania, for sharing with me his hands-on experience on matters like PCB design and electronic circuits behavior.

Many thanks should also be directed to the MHL's graduate and undergraduate fellow students for offering their perspective of things and encouraging me in times of hopelessness or debugging rage. Special thanks are directed to Christos Penoglidis for talking me into this thesis, in the first place.

I should not forget to thank Karol Dobek as well, for cordially guiding me through various Bluetooth documentation and protocol irregularities (and, be assured, there are quite enough of them out there).

Finally, my parents for "being there" for me throughout the whole venture, backing me up and loving me silently when all else had failed and, finally, God -who makes this and all things possible.

1

Introduction

1.1 The Need To Go Wireless

These days, communications and computing systems are subject to a smooth convergence and, denying the fact that this has clearly been predicted several years before, would be unfair to those analytical minds that had foreseen this "unification" coming. Communications have mingled with all aspects of human life in ways not easily discerned or -even- taken into consideration. The technology boost in the area of computers has given communications the breathing room they needed to unlock all their potential and abilities to change our everyday life to something previously unheard of. With all this unanticipated growth and the miraculous bonding of the whole planet to a "blabbering" unity in permanent contact, communications have nullified distances and made the transmission of data and voice over the globe a quite trivial issue.

Towards this end, a plethora of devices has appeared like mobile phones, laptop computers digital cameras, and PDAs [1], to name a few. However, this newly-obtained connectivity has made planet Earth a more complicated place to live; until recently, enabling all of these devices to communicate with each other has become cumbersome, often involving the use of special cables (fig. 1.1) to connect the devices together along with device-specific software that might use proprietary protocols. Trying to exchange information among all of these personal devices -in order to achieve the above mentioned connectivity-, a person might need to carry as many cables as devices and still lack assurance that all the devices could interconnect. The inability to share

Fig. 1.1: Various types of cables, all incompatible among them

[1] PDA: Personal Digital Assistant; a.k.a. Palmtop PC

information among devices or the difficulty in doing so, limits their usefulness [1].

1.2 The Bluetooth solution

One robust answer to such problems came with the new wireless technology called "Bluetooth". Bluetooth wireless technology is a short-range radio technology developed by Ericsson and other companies which makes it possible to transmit signals over short distances between telephones, computers and other devices. Bluetooth wireless technology has simplified both communication and synchronization between devices. It has replaced many of the proprietary cables used in the home and office to connect devices together: telephones, printers, PDA's, desktop and laptop computers, fax machines, keyboards, joysticks - almost any digital device that exploits this new technology is able to take advantage of this "wireless feature". More than a mere *replacement for cables*, Bluetooth wireless technology has provided a universal bridge to existing data networks, a peripheral interface, and a *mechanism to form small private ad hoc groupings of connected devices* away from fixed network infrastructures [31]. Such a typical grouping - a PAN[2] – is depicted in figure 1.2.

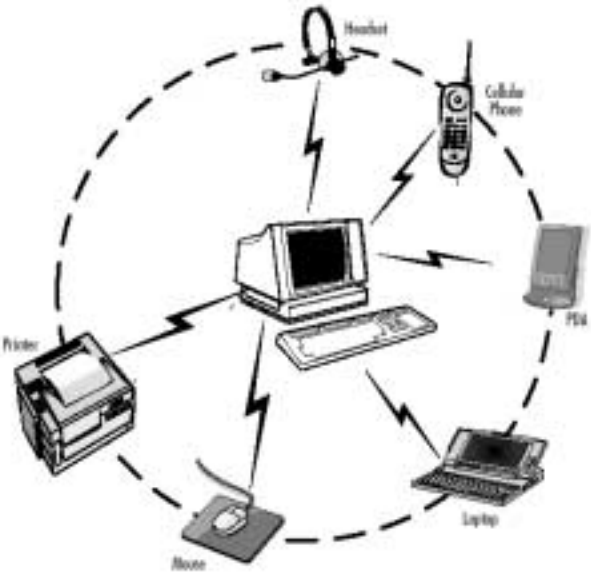


Fig. 1.2: The Bluetooth interoperability Concept [2]

Although an in-depth analysis of the Bluetooth technical characteristics will appear in a following chapter, it must -once more- be stressed out that Bluetooth wireless communication is, first and foremost, a means for cable replacement while, at the same time, it has the capacity of enabling a fully qualified and documented WLAN [3] protocol. This dual role is driven and further advanced by the Bluetooth SIG [4] (SIG, hereafter). The SIG has released a **Bluetooth Core Specification Book**[5] (BT

[2] PAN: Personal Area Network (like LAN, MAN etc.); a network ranging for a few meters
[3] WLAN: Wireless Local Area Network
[4] SIG: Special Interest Group (populated by Ericsson, Intel, IBM, Nokia, 3Com, Agere Systems, Microsoft, Toshiba and Motorola)

Spec. or simply Specification, hereafter) [3] for explaining the complete functionality and full capabilities of the Bluetooth protocol and also a **Bluetooth Profiles Book** (BT Profiles, hereafter) [4] that includes specific case studies / applications of this new technology, in order to make interoperability among all kinds of devices possible. This feature should be combined with all other key elements Bluetooth stands for, like:

i) **low-power consumption**,
ii) **low cost**,
iii) **small size**,
iv) **world-wide use,** and:
v) **security**.

In a nutshell, the Bluetooth Spec. defines a short range (around 10 meters) or optionally a medium range (around 100 meters) radio link capable of voice or data transmission to a maximum capacity of approx. 720 kilobits per second (kb/s) per channel. Radio frequency operation is in the unlicensed industrial, scientific and medical (ISM) band at 2.40 to 2.48 GHz, using a spread spectrum, frequency hopping, full-duplex signal (through TDMA) at a nominal rate of 1600 hops/sec. The signal hops among 79 frequencies at 1 MHz intervals to give a high degree of interference immunity. RF output is specified as 0 dBm (1 mW) in the 10m-range version and -30 to +20 dBm (100 mW) in the longer range version [5].

1.3 Thesis stimulus, scope, purpose and results

Under Ericsson licensing, the Bluetooth protocol has been adopted by many Bluetooth SIG partner companies and appears now in many implementations, partially in hardware, software and/or firmware –depending on the described part of the protocol. The above-stated Bluetooth key features in conjunction with the promise for easy and completely seamless wireless communications among different types of devices has been the best incentive for the market to embrace -ergo, for the developers to boost- this new technology. Many new end-user products ranging from integral solutions (embedded and peripheral ones) to development kits -all discussed in detail in the next chapter- include integrated Bluetooth modules. This has been the incentive for this thesis also, i.e. to mingle with this limitless protocol, exploit all its abilities and make a smart, new addition to the Bluetooth product list

[5] The first really stable BT spec. was v1.0B but currently (June 2003) the latest version is used, v1.1, which combines v1.0B + Errata and also bears some additions (for a detailed revision history see [3], Appendix I) to the specification. Meanwhile, the Bluetooth specification v2 is underway with many more profiles, among others.

from the standpoint of an embedded application. Of course, this was made possible due to Ericsson's kind donation to the M.H.L. of two development kits consisting of a HW/FW - implemented Bluetooth module and the appropriate accompanying Bluetooth SW cores (all covered extensively in Chapter 4).

In detail, in this thesis we have constructed an ***embedded*** *applications platform* (consisting of hardware and software parts) acting as a host to drive fully each Bluetooth module and is called Bluetooth Applications Environment ("***BlueApplE"***, hereafter). We have developed the system on an Atmel microcontroller which makes all required provisions and takes specific actions to handle the Bluetooth modules. Control of the system has been achieved through specific interfaces, thoroughly described in chapter 5. Through the BlueApplE, the typical and special features of the Bluetooth modules have been exploited in a great extent in order to present all the capabilities of this emerging technology, as pinpointed in the previous subsection. Once this goal has been reached, the effort is directed towards deploying a *specific demo application* on the designed system. Hardware and software additions are made to the core design -the BlueApplE- for this demo application to be built, which is the setup and run of a *wireless transparent UART bridge* (***"BlueBridge"***, hereafter) and has not been chosen randomly. By developing the BlueApplE and BlueBridge, the thesis contributions are:

i) to acquire background information and sufficient know-how of the Bluetooth technology,
ii) to build a design which demonstrates the full power of Bluetooth technology and acts as the sandbox for future Bluetooth-enabled designs (BlueApplE),
iii) to provide external and built-in means of validating and testing the proper functionality of the overall design, and
iv) to actually build and present a specific Bluetooth application (BlueBridge) which can be incorporated in many existing devices.

Key elements of this thesis are: i) the **portability**, ii) the **reconfigurability / flexibility**, and iii) the **expandability** of the design (hardware and software), elements that will be stressed out and respected throughout the unfolding of this document. Moreover, in combination with the cost efficiency of the Bluetooth technology, an additional element sought has been **low implementation cost** which has been pursued by using inexpensive components to build the system; yet, this last element has been strictly subject to the descending priority queue: i) component availability, ii) maximum system performance and iii) low cost.

1.4 Thesis roadmap

The contents of this thesis are structured as follows:

Chapter 2 is occupied with the Bluetooth market at the time being. Various products are discussed and their respective producer-companies; also, the functionality, abilities, competitiveness and efficiency of these products.

Chapter 3 takes some time to make an in-depth tour to the Bluetooth protocol inquiring the general guidelines of the technology, the details of the protocol's various layers and an even more thorough examination of the parts of it that are utilized in this thesis. In the end, it attempts to make a brief comparison between Bluetooth and 802.11 as well as Bluetooth and IR.

Chapter 4 guides through the "Applications & Training Toolkit" specific features, its abilities and its limitations.

In *Chapter 5*, the details of BlueApplE will be analyzed, including system architecture, hardware and software issues.

Chapter 6 describes in detail the BlueBridge application and the required setup steps and, at the end, lays proof of its unhindered operation.

The results of such an application are cited in *Chapter 7*, and a brief presentation of the PCBs designed for incorporating both the BlueApplE system and BlueBridge application follows. The chapter ends with a discussion regarding system design cost and overall system power consumption.

Finally, *Chapter 8* covers all the conclusions that can be extracted from this thesis and the future work that can be done to upgrade the BlueApplE system and the BlueBridge application.

2

Relative Research

2.1 Bluetooth products: the billion-unit market

Bluetooth wireless communication has attracted considerable attention since the formation of its SIG was announced in May 1998. With all the hype surrounding the technology, many consumers, analysts, and others were ready to see real products in the marketplace. Fortunately, beginning in 2000, the first products started to appear like notebook computers with Bluetooth wireless technology, PC cards for existing notebook computers, mobile phones, HIDs[6], wireless headsets and network access points, to name a few. While 2001 will be remembered as the year the Bluetooth marketplace began to take off, till now (June 2003) the number of Bluetooth-enabled products has reached a six-digit order of magnitude. More surprisingly -in BBC words:

> "Despite the aura of gloom pervading the computer market in 2002, one technology looks like it had a good year. A report out this week shows that Bluetooth, the short-range radio system, experienced dramatic growth over the last 12 months.", Thursday, January 16, 2003

By all accounts (and by many research institutes) Bluetooth products are expected to skyrocket in the following years building a billion-unit market. Many important industry "players" like Siemens, Philips, CSR, Sun, Hewlett Packard, Xilinx and Atmel have accepted the challenge including -of course- all the SIG[7] members: 3COM, Ericsson, IBM, Intel, Microsoft, Motorola, Nokia and Toshiba. Also taking into account the "Associate" members[8] and "Adopter" members[9], the incredible number of over 2.400 companies comes up. An interest for Bluetooth wireless technology of this amplitude makes this so-called billion-unit market far more probable and -at the same time- leads to a miniaturization of production cost towards the all famous $5-cost goal for implementing the Bluetooth wireless technology (RF and Baseband part only). All this can be easily anticipated from a typical forecast graph conducted

[6] HID: Human Interface Device, e.g. mouse, keyboard, joystick etc.
[7] SIG members also known as "Promoter" members
[8] "Associate" members: companies participating in the BT spec. advancement
[9] "Adopter" members: companies simply making use of the Bluetooth trademark and logo

by Intex Management Services (IMS) for some main product families in the time margin: 2001 – 2005 (fig. 2.1):

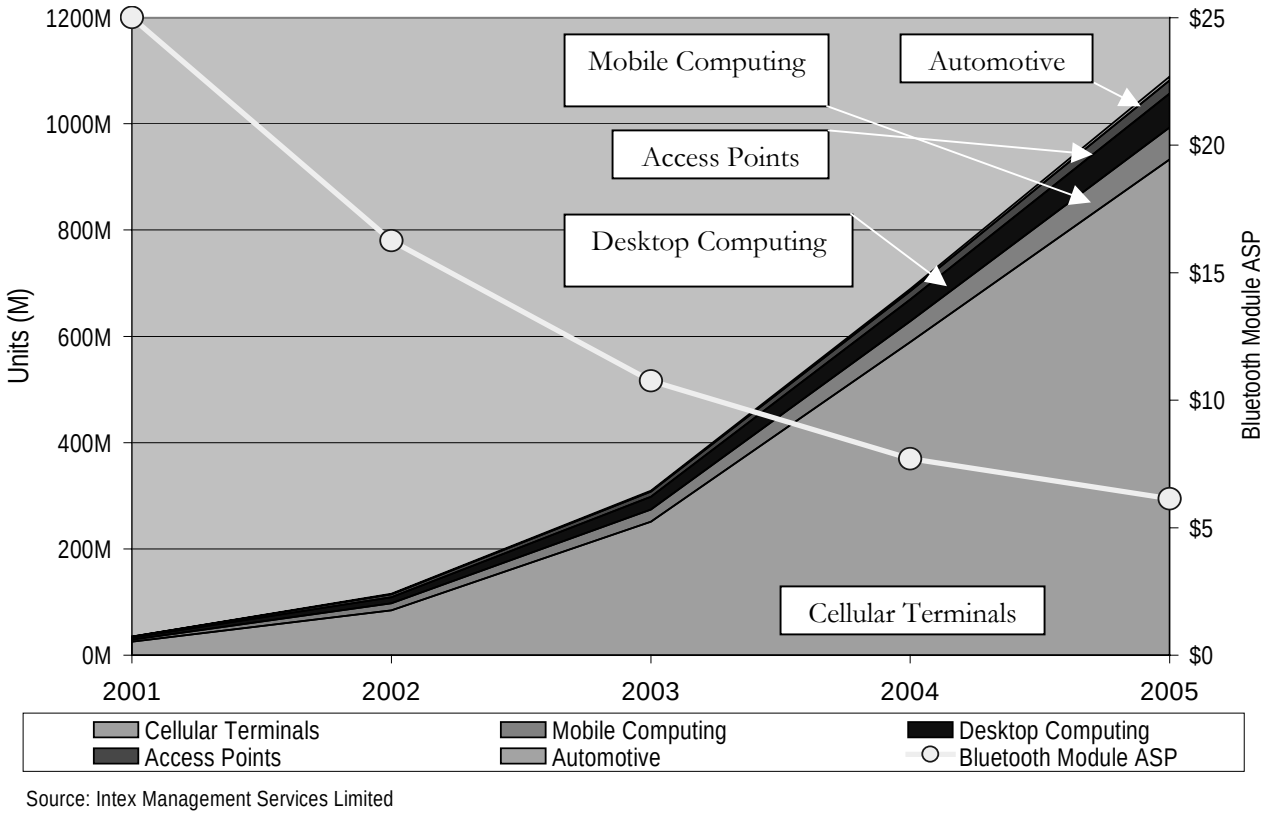


Fig. 2.1: Bluetooth market forecast

2.2 An overview of the Bluetooth protocol

Obviously, now, Bluetooth wireless technology is getting ubiquitous in many fields like the IT, the biomedical and the automotive industry. In an effort to understand what this Bluetooth "black box" that is -after all- incorporated in all Bluetooth-enabled devices is, a brief overview of the structure i.e. of the protocol stack of the Bluetooth technology follows. This stack would be far more simplistic should Bluetooth aim at just replacing the cables between devices. However, as stated in the previous chapter, Bluetooth is also aiming at connecting different types of devices by forming PANs. To allow for this so-called interoperability between devices, the Bluetooth technology is built on a carefully defined protocol stack (see a rough sketch in fig. 2.2). Whereas the BT spec. does not define what part of the stack should be implemented in HW or SW, Ericsson's early interpretation -a proven stable architecture-

draws this line in the "Host Controller Interface" (HCI, for short) separating the Host Controller (transport layers) from the Host (middleware & application layers), as seen in figure 2.2. Members of the transport group are the Radio, Baseband and LMP whereas members of the MW group are the remaining layers.

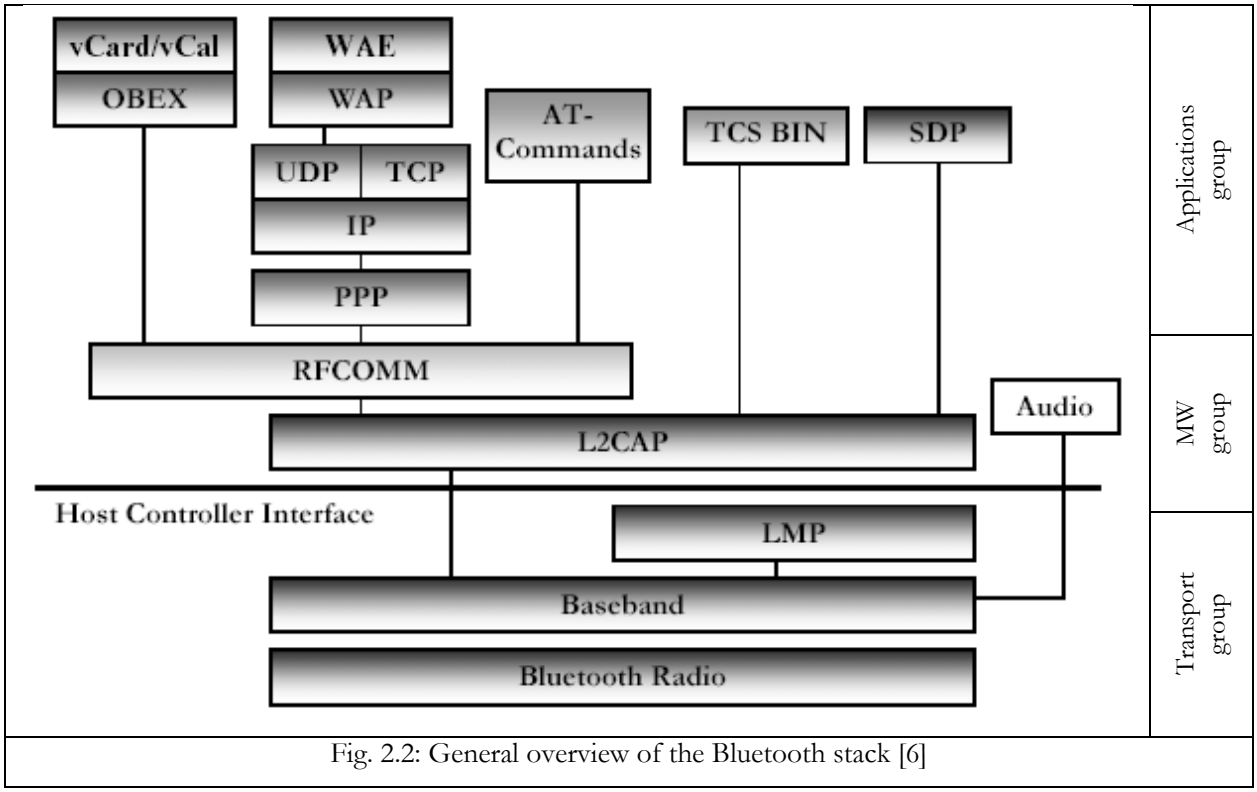


Fig. 2.2: General overview of the Bluetooth stack [6]

Conforming to the purpose of this section, a more detailed description of the various layers of the stack is left for the following chapter where the whole Bluetooth architecture is presented. Suffice to say that this HW / SW separation in the HCI layer is not mandatory but is very popular and indeed respected in the products of the majority of manufactures as it provides a stable, flexible and -usually- cost-effective solution. However, the final decision is up to the manufacturer / designer depending on the needs of the system under development.

Furthermore, the host side of the protocol stack (MW and/or any application layers) can be implemented either as an integral part of the rest of the stack (residing in the Baseband CPU) -i.e. one chip is utilized- or as a SW/FW running inside a separate host CPU (e.g. PC CPU, microcontroller, embedded processor) –i.e. two chips are utilized- (fig. 2.3). The 2-chip implementation is known as the

2-CPU solution and its advantage over the 1-chip implementation (also known as 1-CPU solution) is that it offloads the host CPU (e.g. PC CPU) of all real-time Bluetooth activity and cuts the host CPU overhead to a minimum. Its drawback, however, is that it is less cost- and space-efficient a solution. These two solutions are the main approaches of the companies to the Bluetooth products depending on the special features pursued, like mobility, low power, low overhead (2-CPU solution) or like low cost, portability, performance (1-CPU solution) and largely explain the rationale behind the design of Bluetooth products presented below. In time, however, given the boost in power of the embedded systems, the technology trend favors the 1-CPU implementation (e.g. mobile phones, PDAs).

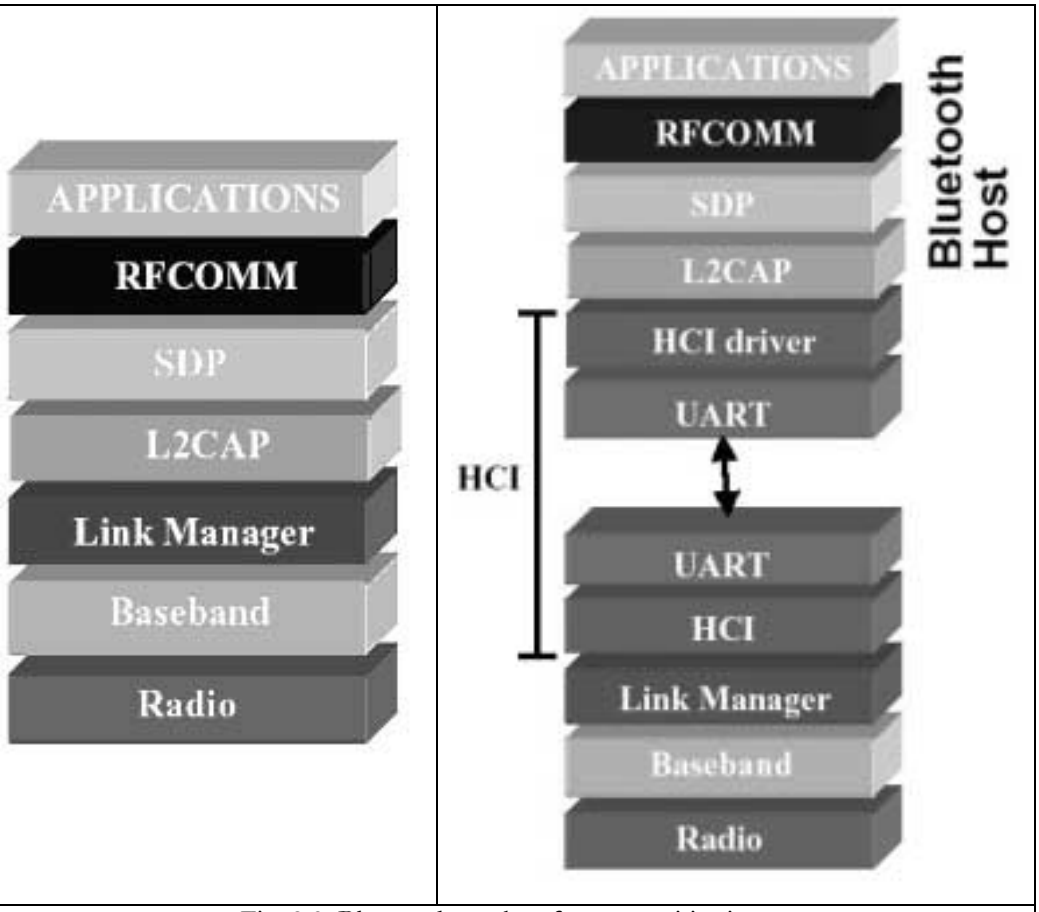


Fig. 2.3: Bluetooth stack software partitioning

The complexity of the Bluetooth protocol stack has led to diverse implementations ranging from single components / modules mainly implementing the transport layers (below the HCI) in HW or even in FW (residing in on-chip or off-chip ROMs) to complete end-user products like the ones commercially

advertised: wireless access points, wireless PCMCIA cards, as well as Bluetooth-enabled PDAs, cell phones, HIDs, printers, headsets etc.. Not surprisingly, developer kits were among the first products available, since these hasten the development of more Bluetooth-enabled applications. In an attempt to filter the bulk of information regarding all the Bluetooth-enabled products currently available, the framework of the research conducted here will be restricted to specific product lists. Since this thesis addresses the abilities of a specific development kit (examined in chapter 4), more attention will be drawn to those competing kits already in shipment around the world. Also, some typical or special products utilizing the Bluetooth technology will also be presented. Finally, this research will encompass some singular Bluetooth components not intended for end-users but -rather- for designers. A tree, roughly outlining the extent of Bluetooth-enabled products, appears in fig. 2.4.

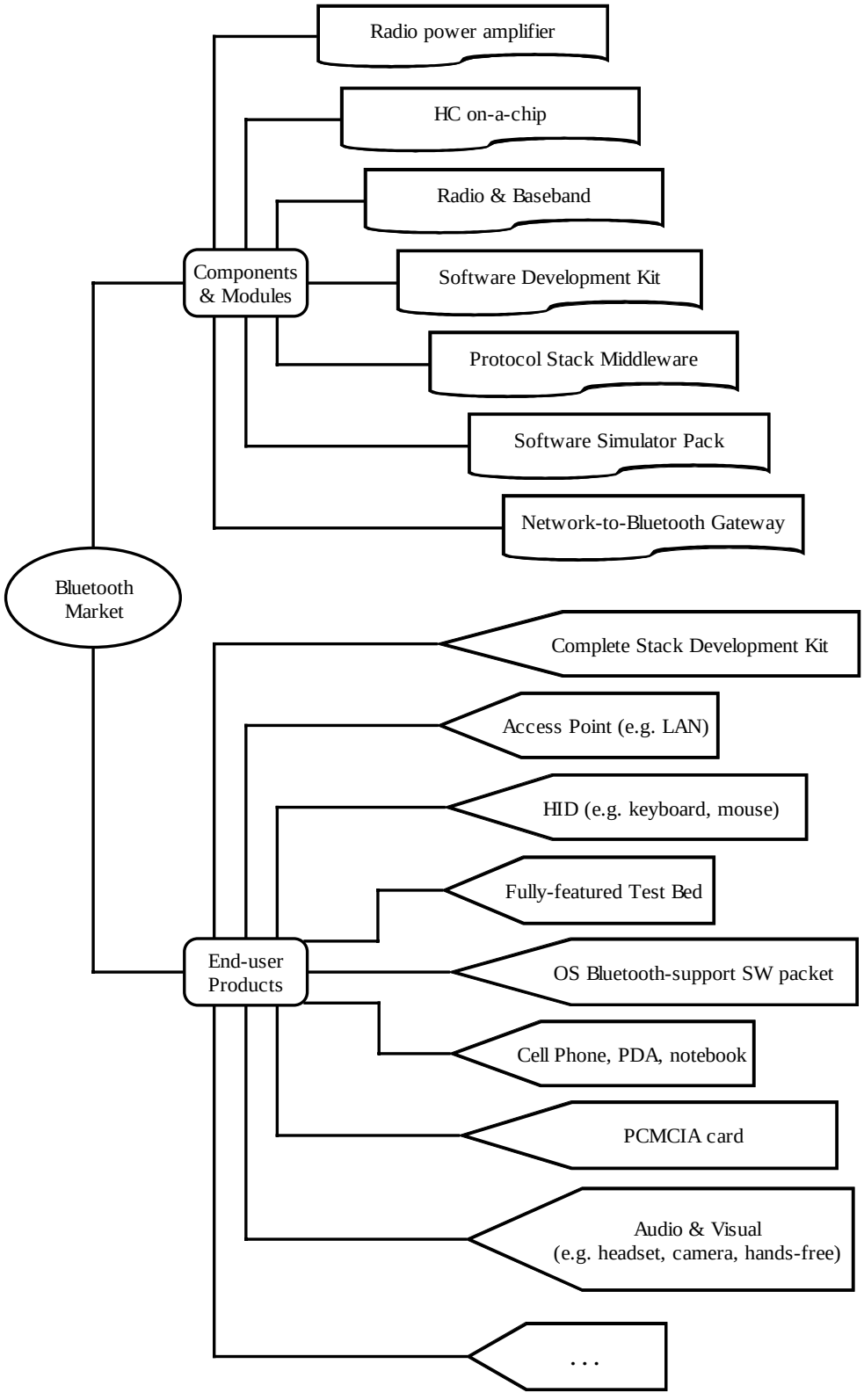


Fig. 2.4: Current Bluetooth product tree (first half of 2003)

2.3 Components & modules

Especially during the past two years many companies have "gotten their hands dirty" with the Bluetooth technology. Long before full products hit the market, many separate building blocks have appeared and keep appearing, creating a circle of creativity in the sense that such components, at first driven by the Bluetooth motto: "throw the wires away", helped to further expand its market share and -most importantly- the time-to-market by providing Ericsson-qualified[10] parts of the Bluetooth stack (in SW, HW or FW) and compliant with the Specification.

2.3.1 Atmel: power amplifier & front-end IC

Starting with three rather low-level but very substantial for the Bluetooth technology products from Atmel:

i) The T7023 is a monolithic SiGe power amplifier and is especially designed for operation in TDMA systems like Bluetooth and WDCT. Model T7025 is also a monolithic SiGe power amplifier and is aimed at operation in TDMA systems like Bluetooth, Home RF and ISM proprietary radios. Of course, the two chips are compliant with the ISM band (2.4 GHz).

ii) The T7024 is a monolithic SiGe 20 dBm RF transmit/receive front-end IC with power amplifier, low-noise amplifier and T/R switch driver and designed also for TDMA systems like Bluetooth and WDCT. The chip features 23 dBm POUT typ., low noise (2.0 dB typ.), high gain and ramp controlled output. This 20 dBm boost expands the Bluetooth range beyond 100 m. A block diagram of T7025 is depicted in figure 2.4.

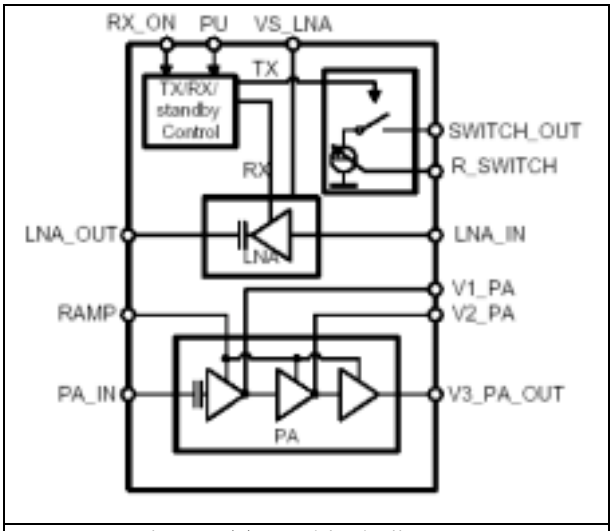


Fig. 2.4: T7025 block diagram:
Atmel Bluetooth power amplifier

[10] From the early beginning Ericsson -as a leading member of the Bluetooth SIG- has established the "Bluetooth Qualification Review Board" (BQRB) which is responsible for checking all new products bearing the Bluetooth logo for compliance with the BT spec. and interoperability. In the opposite case, the Bluetooth-enabled product ought to undergo one or more revision steps before entering the market. (For more details, see [8], [9])

2.3.2 Atmel: Single Chip Bluetooth Controller

From another perspective, Atmel also offers a single chip Bluetooth Controller providing the functionality for high data rate, short distance wireless communications in the free ISM band. In conjunction with a 2.4 GHz transceiver, it provides a cost effective networking solution for a wide range of digital communication devices and computer peripherals (fig. 2.5). Integration is simplified due to the incorporation of three different interfaces: USB and 16550 UART compatible interfaces and a PCMCIA interface conforming to the PC Card 95 specification. Additionally, a voice coding / decoding module is provided. The AT76C551 is comprised of a baseband processor. This processor carries out all bit-level processing after modulation / demodulation of the Bluetooth bit-stream. It controls the transceiver and dedicated voice coding/decoding. The AT76C551 has an ARM7TDMI processor core with support for internal and external memory, as well as the interface core logic. The powerful RISC processor in the ARM7TDMI carries out all but the low level baseband functions.

Concisely, its features are:

- ✓ Implements Bluetooth Specification on Short Distance Wireless Communication in 2.4 GHz ISM Band
- ✓ Provides 1 Mbps Aggregate Bit Rate
- ✓ Supports Frequency Hopping Spread Spectrum Physical-layer Interface to Dedicated Transceiver with Frequency Hopping Algorithm Implemented in Hardware
- ✓ Provides Baseband Functions in Hardware which Implement Bluetooth Low-level Bit Processing Such as Forward Error Correction (FEC), Header Error Check (HEC) and CRC Generation/Checking and Encryption/Decryption
- ✓ Integrated ARM7TDMI RISC Processor
- ✓ Glueless SRAM Interface, Supporting Up to 256K Bytes of Memory
- ✓ Glueless Flash Memory Interface, Supporting Up to 256K Bytes of Nonvolatile Memory
- ✓ Glueless PCMCIA Bus Interface Conforming to PC Card Standard – Feb. 1995
- ✓ USB Interface Conforming to Universal Serial Bus Standard Version 1.1
- ✓ 16550 UART Core Offering 32-byte Receive FIFO and Programmable Baud Rate
- ✓ Programmable 8/16-bit Wide External Memory Interface
- ✓ Supports Multiple Reference Clock Frequencies (13.000, 14.400, 16.800, 19.440 MHz)
- ✓ 176-lead LQFP
- ✓ 3.3V Supply

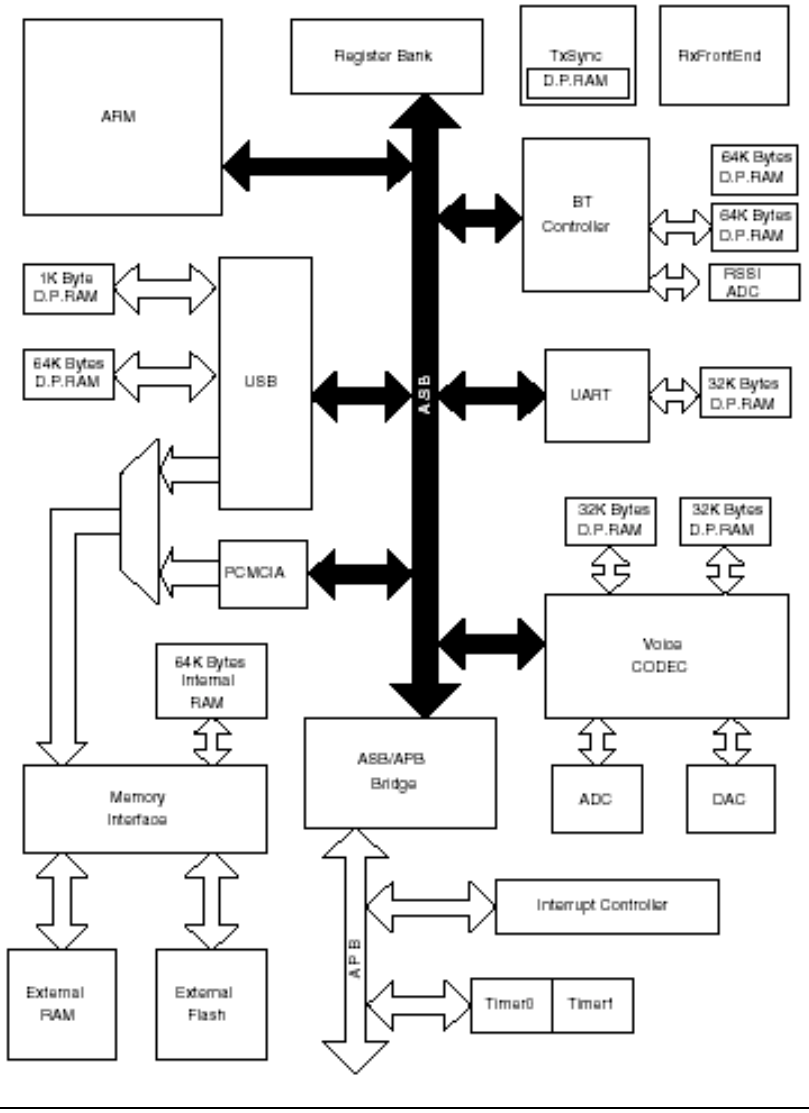


Fig. 2.5: AT76C551 block diagram,
Atmel Bluetooth controller

2.3.3 Infineon Technologies: Baseband Controller

At a first level, Infineon Technologies is offering two versions of the implementation of a Baseband controller chip:

i) "BlueMoon Single - PMB8760" for handheld applications. This Single Chip architecture integrates a high-performance CMOS radio component in the Bluetooth Version 1.1 qualified and proven baseband controller featuring:

- ✓ Integrated ROM for lowest cost
- ✓ External flash for development and production ramp-up

- ✓ Integrated 13 MHz low power oscillator with on-chip PLL
- ✓ Programmable power-down mode
- ✓ 13 or 26 MHz clock input for joint use with GSM clocks (optional)
- ✓ Full SW tuning and trimming (no manual tuning points)
- ✓ State-of-the-art CMOS technology
- ✓ High RF sensitivity (-85 dBm @ 0.1 % BER)
- ✓ On-chip 2.5 GHz RF driver amplifier with +4dBm output power

The block diagram of PMB8760 is depicted in figure 2.6.

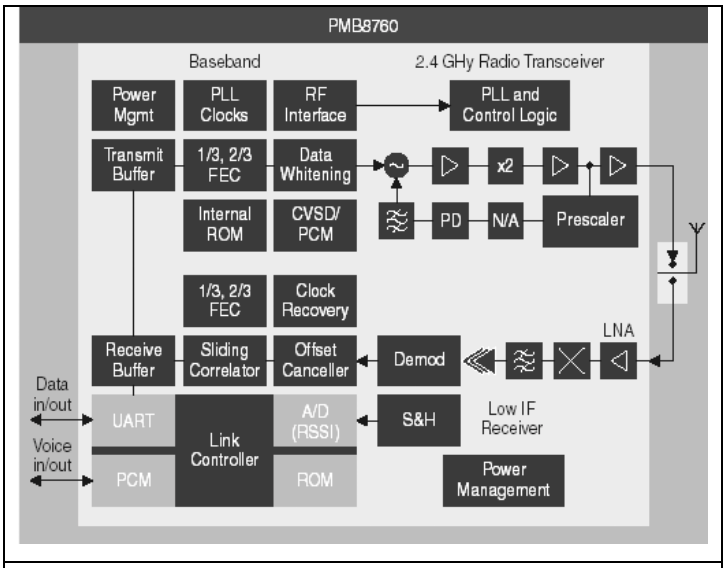


Fig. 2.6: PMB8760 block diagram, Infineon Tech.

ii) "BlueMoon Single Cellular - PMB8761", which essentially uses the Single Chip architecture optimized for the application in cellular phones. Highlights are: reduced power consumption and minimized PCB space. It is ready for Bluetooth Version 1.2 qualification. Its features are:

- ✓ Bluetooth 1.2 support (Adaptive frequency hopping, Extended SCO, Fast connection setup)
- ✓ Integrated ROM for lowest cost
- ✓ Programmable power-down mode
- ✓ Clock input for joint use with GSM/CDMA/3G clocks (11.5 to 46 MHz)
- ✓ Full automatic tuning and trimming (no manual or SW tuning)
- ✓ High RF sensitivity (-85 dBm @ 0.1 % BER)

- ✓ On-chip 2.5 GHz RF driver amplifier with max. +7dBm output power
- ✓ Automatic control of output power for reduced power consumption
- ✓ Compatible to available GSM/GPRS/CDMA baseband solutions

The block diagram of PMB8754 is depicted in figure 2.7.

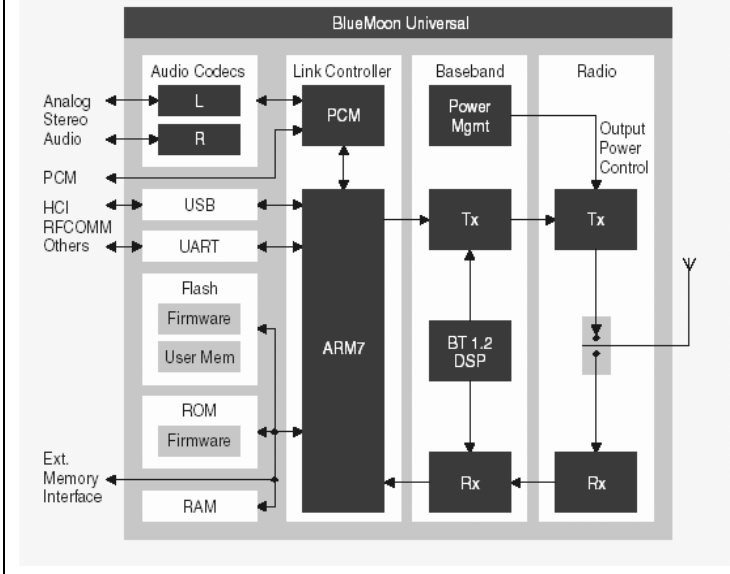


Fig. 2.7: PMB8754 block diagram, Infineon Tech.

2.3.4 Infineon Technologies: Bluetooth chip

Designed for the PC segment, the "BlueMoon UniUSB - PMB8754" product provides a USB interface and embedded flash memory for embedded application specific software. An open software API allows for application specific ARM7 programming. The chip features are:

- ✓ ARM7 scalable up to 78 MHz
- ✓ 0.13µm CMOS
- ✓ Bluetooth 1.2 Support
- ✓ Adaptive frequency hopping
- ✓ Extended SCO
- ✓ Fast connection setup
- ✓ Embedded FLASH
- ✓ Lowest power consumption

- ✓ <40 mA peak in active mode
- ✓ <20 μA in low power mode
- ✓ 1.8 V supply voltage
- ✓ Programmable I/O voltage domains (1.5 ... 3.6V)
- ✓ Dynamic control of output power

Its block diagram is the one appearing in figure 2.7.

Likewise, the company also releases "BlueMoon UniCellular - PMB8752" which is optimized for Cellular Phone applications great cost efficiency, lowest power consumption and the same features of the PC-version chip.

2.3.5 Motorola BTPLATFORM: Motorola Platform Solution

To enable Bluetooth product developers to meet increasingly aggressive time-to-market requirements, Motorola offers a complete and comprehensive RF-to-applications Bluetooth platform solution with the Bluetooth Platform Solution featuring:

- ✓ Coexistence with 802.11b
- ✓ Robust RF performance
- ✓ Superior voice
- ✓ Interoperability
- ✓ Power advantage
- ✓ Two-chip architecture

The platform chipset provides a low-power, low-cost, single-supplier system solution. The key components that compose the chipset are:

- ✓ MC71000: Bluetooth Baseband Controller IC
- ✓ MC13180: Bluetooth Low-Power Wireless Data Transceiver
- ✓ MC72000: Integrated Bluetooth Radio that combines the MC71000 and MC13180 into a single package
- ✓ MMM7400: Bluetooth Low-Power Data Transceiver Module
- ✓ MC13181 (optional): Wireless Power Management IC for Headset and Phone Accessory Applications
- ✓ MRFIC2408 (optional): External Power Amplifier IC for Class 1 Applications

Figure 2.8 gives a sample circuit for utilizing the Motorola chipset:

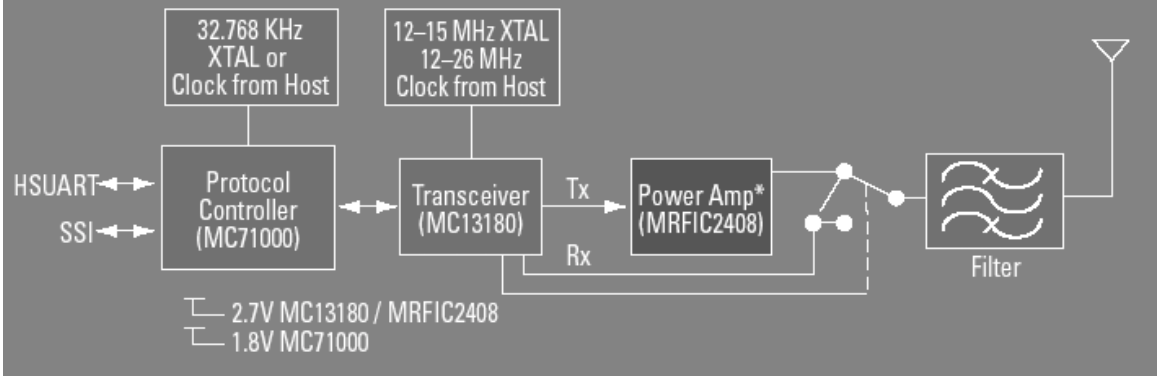


Fig. 2.8: Sample two-chip Bluetooth solution, Motorola

2.3.6 CSR: BlueLab, Professional SDK

CSR has created the BlueLab software development kit (SDK) to support Bluetooth projects, which comes with ready-to-use (or adapt) source-code FW for specific applications. Tried and tested example HW designs to accompany this application software are also available free, providing an exceptionally fast and low-risk path to market for OEMs.

BlueLab is based on the popular C language for flexibility - using the proven GNU C compiler. The SW allows users to create on-chip ANSI/ISO C programs in BlueCore's (see below) user space, while maintaining the integrity of the pre-certified Bluetooth protocol stack thanks to the chip's virtual machine environment (which can save many weeks of development time, and avoid huge time delays and costs in compliance testing).

BlueLab provides the basic SDK with the C compiler, graphical debugger, download tools and utilities, plus source-code application software for Headset and Hands-Free profiles, and various utilities.

2.3.7 CSR: BlueCore2-External, Single Chip Bluetooth Solution

Also from CSR comes BlueCore2-External which is the counterpart to the BlueLab SDK: a single chip radio and baseband IC for Bluetooth 2.4GHz systems. It is implemented in 0.18μm CMOS technology. When used with external flash containing the CSR Bluetooth software stack, it provides a fully

compliant Bluetooth system for data and voice communications. A block diagram is available in figure 2.9.

BlueCore2-External contains a 16-bit RISC microcontroller which is used to run the Bluetooth protocol stack. CSR has implemented a Virtual Machine (VM) on this RISC microcontroller. This provides a "sand box" in which applications can be run without adversely affecting the underlying Bluetooth communications stack. CSR's BlueLab SDK (discussed above) contains tools for compiling and debugging VM applications.

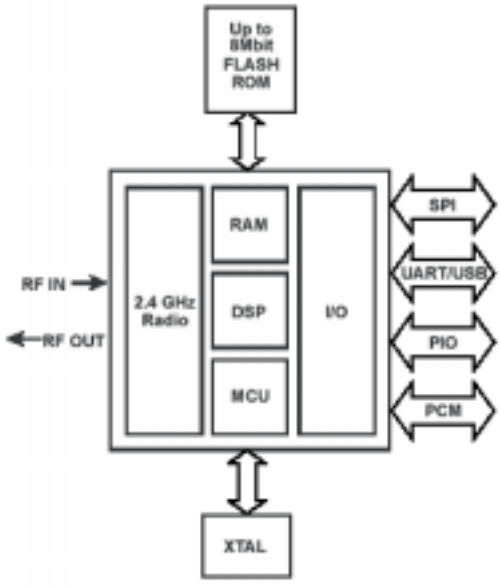


Fig. 2.9: BlueCore2-External block diagram, CSR

2.3.8 Teleca Comtec: CAN[11]-to-Bluetooth Gateway

Following the growing number of application areas for Bluetooth in cars and trucks, Teleca has developed a CAN to Bluetooth gateway aiming at the automotive and telematics industry. The gateway is built on state-of-the-art microcontroller technology, with a modular design and a generic platform, adaptable to a variety of application areas:

- ✓ Automotive – CARB, unwired diagnostics, production supervision, ECU prototyping
- ✓ Medical – gateway to external units
- ✓ Industry – control of mobile units

Based on a modularized HW and SW platform, the gateway can easily be modified, in SW only, to an industrialized solution for a specific purpose. This CAN-to-Bluetooth Gateway is built on two integrated boards – a motherboard (CAN) and daughter board (Bluetooth) – that can also be used as stand-alone units. Its features are:

- ✓ Two CAN channels, supporting speeds up to 1Mbit/s
- ✓ J1708 channel

[11] CAN: Controller Area Network

- ✓ K-line and L-line channel
- ✓ Serial data interface
- ✓ OBD-II compliant
- ✓ Bluetooth Serial Port Profile, possible for any Bluetooth-enabled device to connect to the gateway
- ✓ Flexible Bluetooth to CAN protocol – no limitations to the CAN protocol and configurable to utilize the Bluetooth bandwidth.

2.3.9 IBM alphaWorks: MW for Bluetooth wireless devices, BlueDrekar

BlueDrekar protocol driver is IBM's MW based on Bluetooth specifications allowing Bluetooth wireless devices, ranging from phones to household appliances, to communicate reliably with each other. BlueDrekar includes the following:

- ✓ Three loadable modules: "btstack.o", "sdp.o", and "rfcomm.o", including support for layers from the Host Controller Interface (HCI) to RFCOMM and the Service Discovery Protocol (SDP) layer
- ✓ Manual pages and an open application programming interface (API)
- ✓ The executable of the bdd daemon, which is used to provide support for SDP
- ✓ Makefiles, documentation files, and sample programs that implement various Bluetooth profiles. These files may be used to help write applications.

BlueDrekar can be used with various HCI transport layers and -ergo- provides a very flexible simulation and implementation tool.

2.3.10 IBM: Bluetooth ad-hoc network simulator, BlueHoc

BlueHoc is an open source Bluetooth technology simulator. Released under IBM public license it allows users to evaluate how Bluetooth performs under various ad-hoc networking scenarios. The key issues addressed by the simulator are:

- ✓ Device Discovery performance of Bluetooth
- ✓ Connection Establishment and QoS negotiation
- ✓ Medium access control scheduling schemes
- ✓ Radio characteristics of Bluetooth system
- ✓ Statistical modeling of the indoor wireless channel
- ✓ Performance of TCP/IP based applications over Bluetooth

The following Bluetooth layers have been simulated:

- ✓ Bluetooth radio
- ✓ Bluetooth baseband (BB)
- ✓ Link Manager Protocol (LMP)
- ✓ Logical Link Control and Adaptation Protocol (L2CAP)

BlueHoc is based on the open source simulator "Network Simulator" (NS) and provides a Bluetooth extension to NS. Ergo, it uses the TCP/IP simulations of NS to provide a complete simulation model for Bluetooth performance evaluation. It also provides a simulation platform for testing performance of various routing and service discovery protocols over ad-hoc networks. Though the simulation model of BlueHoc closely approximates Bluetooth protocols, it can be used as a platform to evaluate performance of proposed improvements to the technology, as well.

2.3.11 Synopsys: DesignWare BlueIQ Core

DesignWare BlueIQ Core is a fully integrated Bluetooth baseband controller (BB) and link manager (LM) qualified to the v1.1 Bluetooth specification. It includes a royalty-free, embedded microcontroller and FW to implement the lower layers of the Bluetooth protocol stack, offloading the host processor of all real-time Bluetooth processing. It features:

- ✓ Provides two-CPU architecture for unmatched ease of integration
- ✓ Includes microcontroller (6811-compatible) that offloads host processor of 100 percent of baseband and link manager processing
- ✓ Designed for low power and system cost
- ✓ Simple HCI-over-UART interface that allows connection to any host processor
- ✓ Optimized connection to Silicon Wave SiW1701 radio modem
- ✓ Small gate count and memory footprint
- ✓ User configurable to match application needs as follows:
    - o Up three voice channels and optional PCM
    - o Hardware encryption
    - o Flexible memory interface supports
    - o RAM, ROM, and Flash configurations
- ✓ Bluetooth development kit available for application development on PC or other UART or USB-equipped hardware platform

- ✓ Qualified to Bluetooth v1.1 specification

Although the DesignWare BlueIQ two-CPU architecture is well-suited as is for most Bluetooth applications, certain applications may benefit further using BlueIQ in a one-CPU configuration, attaching the baseband controller directly to the host CPU. To support this, a user needs to remove the embedded 6811-compatible microcontroller and directly connect the BlueIQ Baseband controller and peripherals to the chosen host processor through BlueIQ's APB (AMBA) local bus interface. This connection also requires porting the BlueIQ low-level Bluetooth stack software to the chosen processor. This hardware and software integration is available as a service from Synopsys. The block diagram of BlueIQ appears in figure 2.10.

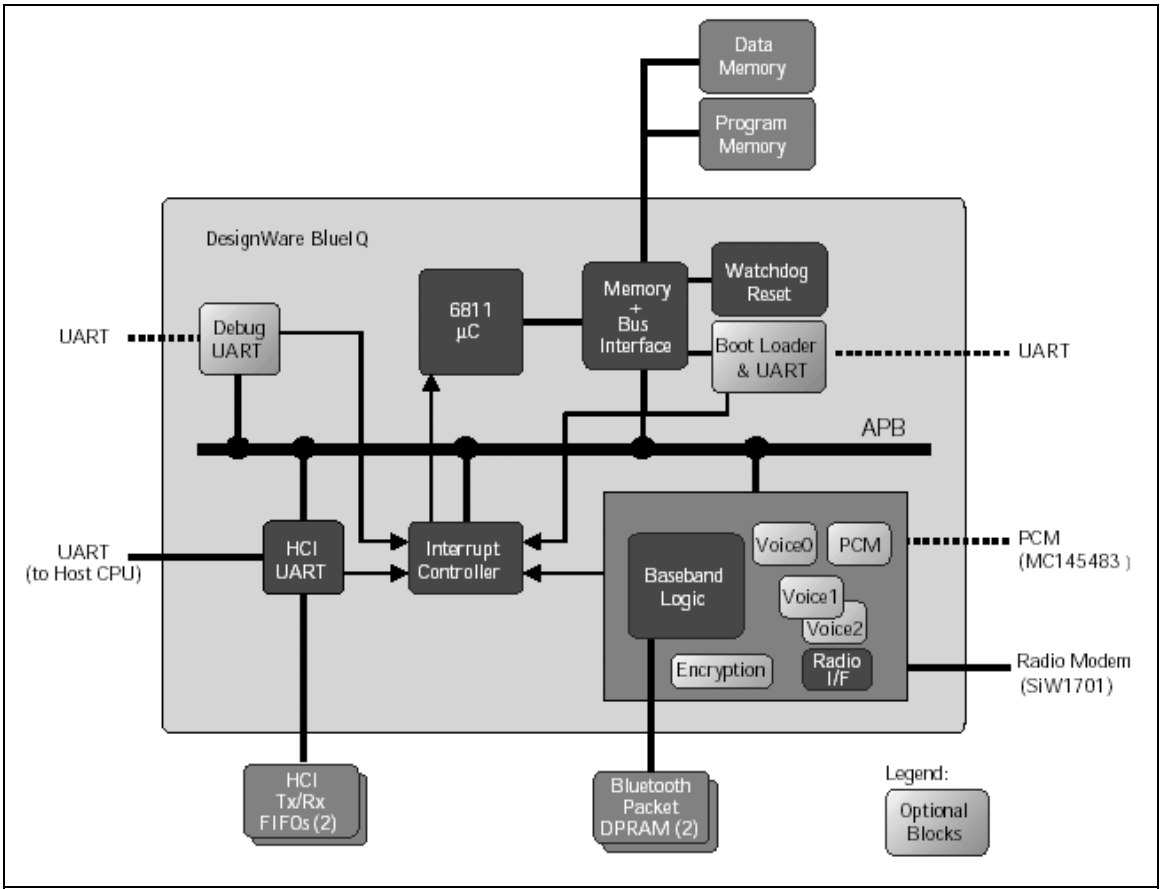


Fig. 2.10: BlueIQ block diagram, Synopsys

To aid in software application development, Synopsys also offers a Bluetooth Development Kit (officially listed with the Bluetooth SIG). The Kit contains the qualified DesignWare BlueIQ Core in silicon and a SiW1701 radio modem. It also includes a demonstration version of Mezoe's Interface

Express incorporating BlueStack product, the industry's most widely adopted high-level Bluetooth stack and profile software.

2.4 End-user products

The SW and HW components presented above are indicative of the picture of the current Bluetooth market. However, this picture would be incomplete should the research end here. Indeed, many companies have "put two and two together" and provide fully-featured, user-friendly end-user products showing what the Bluetooth technology is all about. Even though there has been an effort to gather samples from all important fields of application, special attention has been drawn to development kits due to the nature of this thesis utilizing one such kit; in so doing, comparisons among them can be formed.

2.4.1 Philips: BlueBerry Developer's Kit (BByK)

The Blueberry Developer's Kit (BByK) is a comprehensive tool set that supports hardware and software development for Bluetooth solutions from Philips Semiconductors. It has been built around the Blueberry baseband IC PCF87750, the transceiver IC UAA3558 and the embedded Bluetooth protocol stack. Philips Semiconductors has designed the BByK to support customers creating Bluetooth enabled devices. It is suitable for software development and allows straightforward system validation by offering flexible interfaces to peripheral memories and CPU emulation tools.

A flexible motherboard / daughterboard concept enables development of different combinations of RF and baseband ICs from Philips Semiconductors. The motherboard gives access to debug signals, external memories and in-circuit emulation tools. It has a standardized interface to the daughterboard, which holds the baseband controller. The Blueberry baseband controller is built around the ARM7TDMI core. The daughterboard needs to be plugged on the motherboard for normal operation. The daughterboard or Target Board (TB) holds the Blueberry baseband controller. Three target boards exist containing different versions of the Blueberry chip:

- ✓ TB208: contains the Blueberry baseband chip in LQFP208 package without internal Flash memory
- ✓ TB208E: same as TB208, but dedicated for ARM7TDMI emulation. Connections from Blueberry to emulation JTAG and Lauterbach connectors have been made as short as possible
- ✓ TB81: contains the Blueberry baseband chip in LFBGA81 package with internal Flash memory

The BByK comes with two PC-based software packages which both have an easy-to-use graphical user interface. The first package is the BlueStar software downloader to program the on-chip Flash and external Flash memory via the RS-232 serial interface. The second package is the BlueBird HCI tester software to test the whole embedded software stack. The tool also offers the possibility to verify the link quality. A block diagram of BByK appears in figure 2.11.

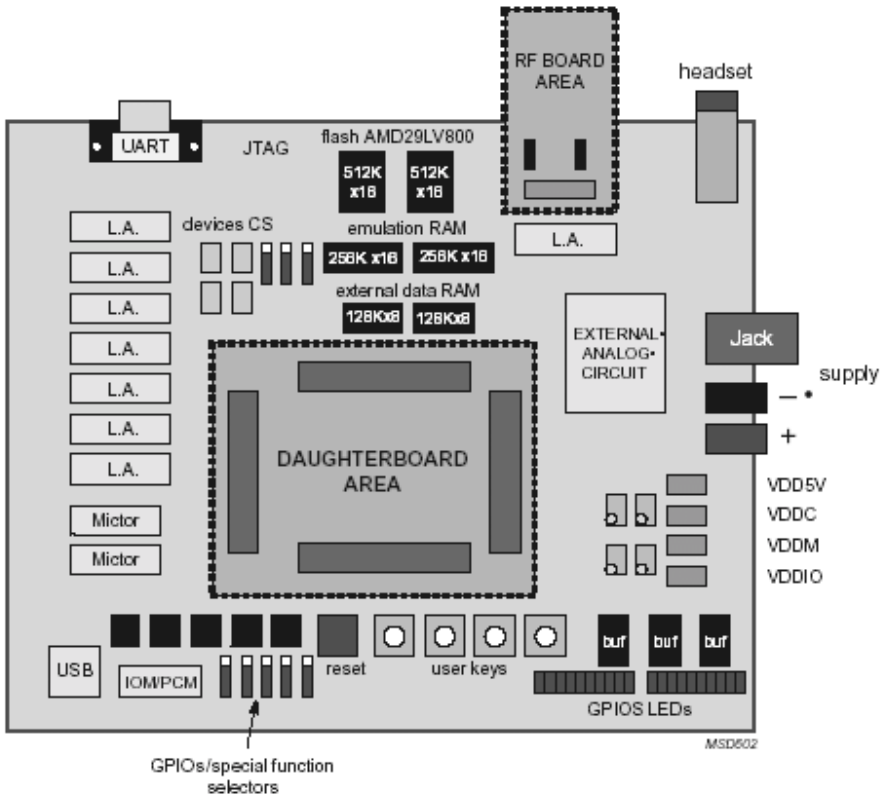


Fig. 2.11: BByK block diagram, Philips

The board's key features are:

- HW:
  - ✓ Fully Bluetooth 1.1 compliant HW and SW
  - ✓ Blueberry PCF87750 baseband controller
  - ✓ Bluetooth RF link based on UAA3558 IC and TrueBlue BGB100 module
  - ✓ 512K x 32 bits on-board Flash memory
  - ✓ 256K x 32 bits on-board emulation RAM
  - ✓ 128K x 16 bits external data RAM
  - ✓ Data and voice links are supported (headset provided)
  - ✓ Connectors for UART, USB, I2C-bus, SPI, IOM/PCM, CODEC, and JTAG interfaces

- SW:
  - ✓ Lauterbach Trace32-FIRE and Hewlett-Packard Real-Time Trace emulator interface
  - ✓ PC based Windows software for internal and external Flash programming
  - ✓ PC based Windows software for HCI testing

2.4.2 IAR Systems: Bluetooth Starter Kit (BSK)

Each Bluetooth Starter Kit (BSK) acts as an autonomous Bluetooth node featuring:

- HW:

  The BSK hardware is connected to the PC via an RS-232 cable or a USB cable. Power is supplied by a main adaptor or via the USB cable. A telephone handset or headset (included in the kit) can be connected to the board to make it a short distance, duplex, radio communication device. The design consists of a main board and a plug-in upgradeable daughter board. The Bluetooth daughter board can be removed and plugged into another (custom) hardware. There is one on-board antenna as well as one antenna connector for an external antenna, for boards that are built into a casing. A switch on the board determines which one to use. An external stub antenna is included. The board also has jumpers for measuring or for re-directing signals to another board, e.g. an embedded CPU board.
- SW:

  A CD-ROM contains all the software and documentation needed to explore and run the kit:
  - ✓ A Windows application that uses the Bluetooth module from Ericsson Microelectronics in various ways
  - ✓ A sample application with source code, which can be used as a starting point for your own prototype development
  - ✓ A USB driver for communicating with the Bluetooth module on Microsoft Windows 98, NT4, NT Embedded and 2000

2.4.3 Symbionics: Ericsson Bluetooth Development Kit (EBDK) / Starter Kit

The EBDK has been designed by Symbionics to enable early adopters of the technology to accelerate the production of prototype applications quickly and easily. It provides a complete and flexible design environment within which engineers can seamlessly integrate the new open wireless standard into a range of digital devices for volume production. The kit bears exceptional-quality scaleable architecture which demonstrates the core features of Bluetooth technology. A variety of interfaces allow for quick development of final applications. The EBDK is designed to meet the needs of the first-time Bluetooth developer and user.

Under Ericsson Licensing the EBDK is available by the following companies:

- 7Layers
- Infineon Technologies Wireless Solutions Sweden
- Integrian
- Teleca Comtec

Its features concisely are:

- SW:

  Host Software Win32 C++ application with user interface that includes:

  - ✓ Bluetooth PC Reference Stack in executable form
  - ✓ Demonstrations of voice/data applications
  - ✓ Bit-error rate (BER) test software
  - ✓ Baseband to Baseband connection support software
  - ✓ Application Wizard for host application development
  - ✓ Application Wizard Demo
  - ✓ HCI scripting tool and sample scripts
  - ✓ Support for both USB & UART
  - ✓ Packet Builder utility for displaying user entered packet details
  - ✓ LMP (Link Manager Protocol) signaling trace

  Baseband Target Firmware:

  - ✓ Basic test FW that allows all the interfaces to be tested
  - ✓ Ericsson's Link Layer and HCI FW that handles the Bluetooth radio packet protocol and provides an interface to the controlling host CPU

- HW:

  Interfaces:

  - ✓ Radio: 0 dBm board with internal or external antenna connectors
  - ✓ Serial Ports: Three RS232 ports with 9-pin 90 degree D-type female connectors
  - ✓ Audio: Pin header for a typical handset
  - ✓ Universal Serial Bus: One USB female
  - ✓ CIF (Common Interface) for improved interoperability tests
  - ✓ I2C: One 4-way pin connector
  - ✓ Status LEDs
  - ✓ General-purpose input/output
  - ✓ JTAG Debug port for ARM CPU

- ✓ On/Off/Reset switch

Application Board Interface:

- ✓ Application Board connector (20-bit address, 16-bit data, Control, Linear PCM)

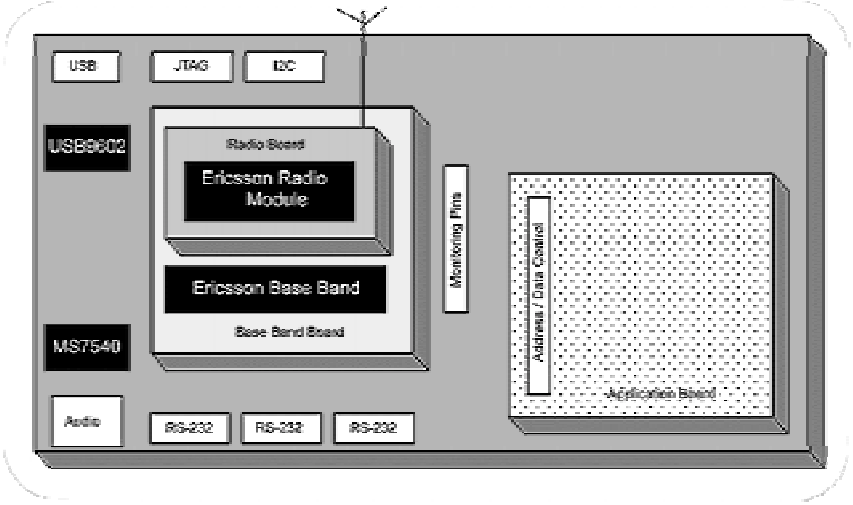


Fig. 2.12: EBDK block diagram, Symbionics

The EBDK block diagram is depicted in figure 2.12. It must be added that the EBDK has been approved as a Blue Unit by the Bluetooth SIG. Blue Units are used in the Bluetooth Qualification Process for qualifying other Bluetooth-enabled products.

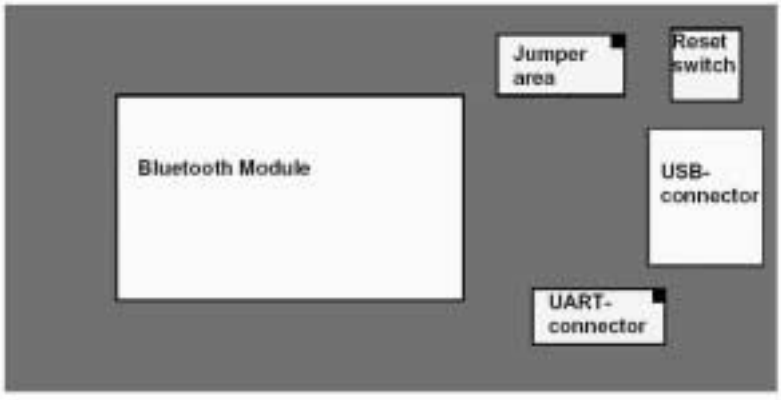


Fig. 2.13: Application Kit block diagram, Teleca Comtec

Under Ericsson Licensing also comes the "Application Toolkit" (fig. 2.13) from Teleca Comtec which is very similar with the one used in this thesis -yet to be examined in chapter 4. The Bluetooth Application Tool Kit features:

- ✓ RF output power class 2 (0dBm-10m range)
- ✓ The Bluetooth Module is compliant with Bluetooth version 1.0b
- ✓ FCC and ETSI approved for RF regulatories
- ✓ 460 kb/s max data rate over UART
- ✓ Multiple interface for different applications:
  - o UART for data
  - o PCM for voice
  - o High-speed USB (v1.1) for data
- ✓ Manual reset possible
- ✓ All the lower layers of the Bluetooth stack included in HW, from HCI down to radio
- ✓ Antenna included
- ✓ Point to Multi-Point Operation
- ✓ Built-in shielding
- ✓ Sideband signals for wakeup and suspend
- ✓ Supports ACPI power management

The module includes software for communication at HCI-level. Included layers are BB, LM and HCI.

2.4.4 Impulsesoft Bluetooth Development Kit (iBDK)

The iBDK acts as a starter, an evaluation or a development kit to enable rapid development of applications that utilize the advantages of Bluetooth wireless technology. It provides an easy-to-use prototyping environment to develop, test and demonstrate Bluetooth applications. The iBDK is a complete solution and provides the flexibility to choose the appropriate combination of data port to Bluetooth. It acts as a bridge between the application specific port and Bluetooth, and the application program can be embedded in the board. The development kit does not require any additional host platform for application execution; the download of applications to the development board can be done directly. Key features of the iBDK are:

- HW:
    - Mainboard:
        - ✓ Processor: 50 MHz ARM7TDMI processor
        - ✓ RAM: SDRAM 4 Mbytes
        - ✓ Flash memory: 2 Mbytes (4 FW banks)
        - ✓ Serial port 1: Bluetooth Module Port
        - ✓ Serial port 2: RS 232 Serial Port
        - ✓ Serial port 3: Debug port
        - ✓ Test and Debug Interface: JTAG debug port for ARM CPU
        - ✓ Serial EEPROM: I2C Based 256 Bytes Configuration Serial EEPROM
    - Bluetooth Daughter Card:
        - ✓ Silicon: Ericsson or CSR
        - ✓ Connectivity: Point-to-multipoint
        - ✓ Optional data port: Ethernet
- SW:
    - Bluetooth: All stack layers (RFCOMM, L2CAP, HCI over UART, SDP)
    - Profile support: SPP, DUN, SDAP, GAP, FAX, LAP, Headset

2.4.5 Motorola: 71000 Bluetooth Development Kit

A three-chip implementation by Motorola (fig. 2.14). The chipset includes:

- MC71000: Bluetooth Baseband Controller IC
- MC13180: Bluetooth Low-Power Wireless Data Transceiver
- MC13181 (optional): Wireless Power Management IC for Headset and Phone Accessory Applications

It has the following features:

- ✓ Connector for mono-audio speaker and microphone (headset application)
- ✓ RS232 interface: Programmable baud rate from 1200 to 921 Kbit
- ✓ UART interface: 5-pin header with RxD, CTS, RTS and GND, 3.3 V signaling, programmable baud rate from 1200 to 921 Kbit, HCI UART transport layer. (The UART and RS232 interfaces cannot operate simultaneously)
- ✓ USB interface: Full speed (12 Mbit/s) USB node device, HCI USB transport layer, 3.3 V operation, self-powered, National USBN9604 USB controller
- ✓ Antenna connector

- ✓ JTAG allowing interface to MC71000
- ✓ Dedicated Bluetooth Audio Signal Processor (BTASP) module included -for high-quality audio- along with on-chip ROM -for low cost (MC71000 features)
- ✓ Low-power, low-cost, high-performance (BiCMOS) flexible transceiver (MC13180 features)
- ✓ Integration of typical functions common in headsets and phone accessory products (MC13181 features)

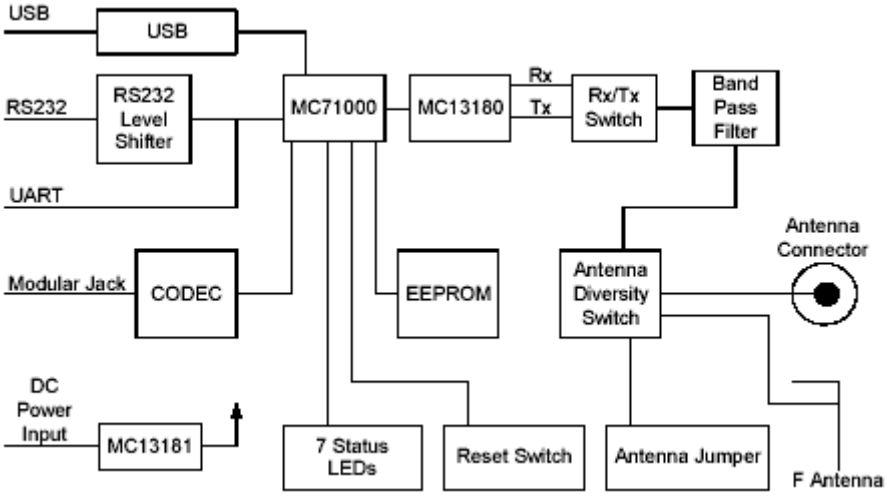


Fig. 2.14: 71000 Bluetooth Development Kit block diagram, Motorola

2.4.6 Agilent: E1852B Bluetooth Test Set

Apart from Development Kits running the Bluetooth stack there are some products that have added features like the monitoring of network electromagnetic characteristics etc.. A solution that combines both is the Test Set from Agilent. It is a low-cost dedicated solution that can establish a link with standard Bluetooth protocol, supporting both Bluetooth normal and test modes. In addition to functionally testing a Bluetooth device, the test set measures key transmitter and receiver characteristics under the conditions defined by the Bluetooth specification. The test set has additional features for development, including demodulated data and clock outputs and can be used as a Bluetooth signal generator or signal analyzer to test elements of the Bluetooth radio.

2.4.7 Wireless Futures, Bluetooth RS-232 solution: BlueWAVE

Due to the nature of the application deployed in this thesis, i.e. the implementation of a wireless UART, some research has been done towards this direction. This and the following product implement just that: a fully-supported RS-232 connection over the air.

The first product is from Wireless Futures and has the following features:

- ✓ Connects directly to existing RS232 interface circuitry (no Bluetooth knowledge required on the host side)
- ✓ All Bluetooth protocol runs inside the module
- ✓ Pin compatible with RS232 circuits to include hardware handshaking
- ✓ Seamless point-point Bluetooth wireless connection
- ✓ Configurable baud rates up to 115kbps
- ✓ 100 meters range (Bluetooth class 1 device)
- ✓ Bluetooth Version 1.1 Compliant
- ✓ Compatible with all Bluetooth 1.1 SPP devices
- ✓ Available as PCB or licensed design

and is the one shown in figure 2.15.

Licensed design

PCB design

Fig. 2.15: BlueWAVE, Wireless Futures

2.4.8 Code Blue Communications, Inc.: Serial Port Adapter (2G)

The 2nd Generation family of Serial Port Adapter (fig. 2.16) enables cable replacement in medical and industrial applications. It allows any device with an RS-232, RS-422, or RS-485 port to communicate wirelessly with no additional software installation.

Features:

- ✓ Low power, with low power modes
- ✓ Faster interface supports full Bluetooth bandwidth
- ✓ ECI interface in all models allows any module to be a multipoint master
- ✓ Longer range: models support Bluetooth Class 1 radio for 100 meter distance
- ✓ Same electrical and mechanical interface for Class 1 and Class 3 OEM modules
- ✓ Available in low-cost plastic packaging for external use
- ✓ More antenna options!

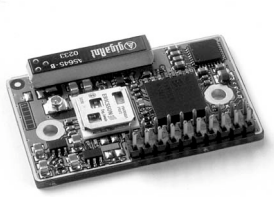


Fig. 2.16: Serial Adapter, Code Blue Comm. Inc.

2.4.9 IBM: WatchPad 1.5

Last but not least, a very interesting application of Bluetooth showing its full potential in embedded low-power, low-cost systems is the WatchPad 1.5 project from IBM (fig. 2.17). Designed to communicate wirelessly with PCs, cell phones and other wireless-enabled devices, the watch has the ability to view condensed email messages and directly receive pager-like messages. In addition, it provides users with calendar, address book and to-do list functions. Future enhancements will include a high-resolution screen and applications that will allow the watch to be used as an access device for various Internet-based services such as up-to-the-minute information about weather, traffic conditions, the stock market, sports results and so on. The watch contains a powerful processor, along with 8 MB of flash memory and another 8 MB of DRAM. Users interact with the watch through a combination of a touch-sensitive screen and a roller wheel. The watch also has both IR and RF wireless connectivity.

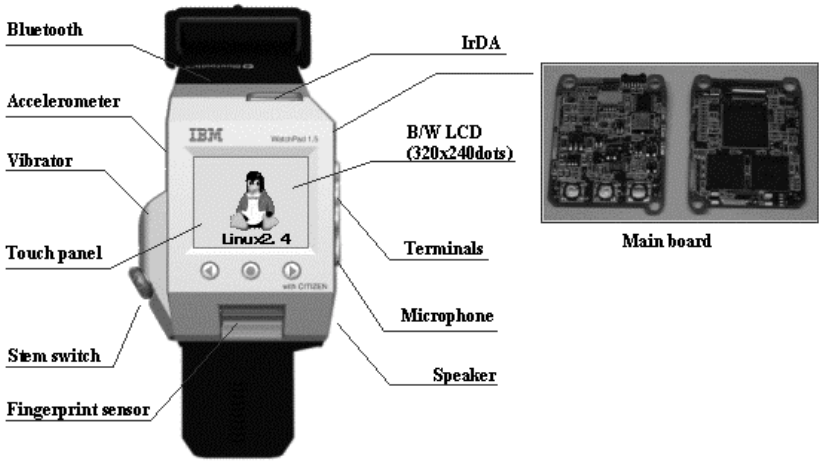


Fig. 2.17: WatchPad 1.5, IBM

The system specifications are:

- HW:
  - ✓ Low-power 32-bit CPU (18-74 MHz)
  - ✓ Low-power DRAM 8MB, Flash 16MB
  - ✓ Bluetooth (V1.1w/voice), IrDA (V1.2), UART (Cradle)
  - ✓ Speaker, Microphone, Fingerprint sensor, Accelerometer, Vibrator
  - ✓ QVGA (320 x 240 dots) reflective monochrome LCD with touch panel
  - ✓ Security feature by adding fingerprint sensor

- SW:
  - ✓ OS: Linux version 2.4
  - ✓ Bluetooth stack: IBM BlueDrekar (L2CAP, SDP, RFCOMM)

2.5 Research conclusions

The list of Bluetooth-enabled products is currently (June 2003) virtually endless. For a more extended list of Bluetooth products, advise the data under the "Product Sheets" list (chapter "References"). From the products examined in this chapter and conforming to the thesis radius, it can be anticipated that a plethora of Bluetooth Development Kits is pushed into the market. Even now, these kits support incredible features that promise to move a design from the scratch paper to the commercial product in no time. This is the so-called time-to-market that all developers wish to reduce at all cost. Such multi-featured kits can indeed minimize the development time by providing complete workstations where issues like module interconnectivity, wiring, voltage levels, mechanical and electromagnetic problems are either inexistent or effortlessly eliminated. Also, the accompanying SW (or FW) packets ensure easier debugging, more flexible designs and a rapid upgrade of the Bluetooth specifications to which the end-user products must comply with no (or little) trouble.

Such a development kit is used also in this thesis, examined thoroughly in chapter 4. As it will be easily anticipated, this kit cannot compete with such development kits as the ones presented above, even by a long shot. It simply integrates the Bluetooth Baseband and RF layer. It communicates with the potential host over a UART or a USB interface and provides no resources for embedded operation (e.g. CPU, FLASH memory etc.) calling for external circuitry and logic to support it. However, it has proven to be more than adequate for the purpose it has been selected; that is, to exploit and make visible the abilities of the Bluetooth technology.

3

The Bluetooth Architecture

3.1 The Bluetooth Name

A rational approach to the Bluetooth architecture would probably start from this mysterious name it bears. *Harald Blatand* was the King of Denmark from approximately 940 to 985 A.D.. According to the history books, in his day King Harald had managed to unite Denmark and Norway and had brought Christianity to Scandinavia. His name, "Blatand", is translated -at least loosely- as "Blue Tooth" because of the folklore about a King's habit to eat blueberries therefore leaving a bluish color on his teeth. In any case, this name had prevailed while the SIG was working on the Bluetooth specification because the technology was intended to unite the computing and telecommunications industries, as well as various companies within those and other industries. However, it was chosen as an internal codename, and -at the time- was not expected to survive as the name used in the commercial arena. While a cool name alone wouldn't be sufficient to sustain the kind of interest that surrounds Bluetooth, it does seem to be one factor that sparks the interest of many. This interest does not appear to be misplaced and -as time elapses- the name appears to be quite fitting since the main goal of the Bluetooth technology has been fulfilled, at least for the time being.

3.2 The Bluetooth Protocol Stack

In the upcoming subsections, a walkthrough of the various layers of the Bluetooth protocol stack is attempted. The approach followed is the one traditionally followed when describing the OSI model12 of network architecture, that is, a bottom-up approach. The stack is conveniently repeated in fig. 3.1. It can be seen that the complete protocol stack comprises of both Bluetooth-specific protocols like LMP and L2CAP, and non-Bluetooth-specific protocols like OBEX (Object Exchange Protocol) and UDP (User Datagram Protocol); those protocols are the ones lightly shaded in fig.3.1. In designing the protocols and the whole protocol stack, the main principle has been to maximize the reuse of existing

[12] OSI model: Open Systems Interconnection reference model

protocols for different purposes at the higher layers, instead of reinventing the wheel. The protocol reuse also helps to adapt legacy applications to work with the Bluetooth technology and to ensure the seamless operation and interoperability of these applications. Thus, many applications already developed by vendors can take immediate advantage of HW and SW systems, which are compliant to the Specification. The Specification is, also, open which makes it possible for vendors to freely implement proprietary or commonly used application protocols on the top of the Bluetooth-specific protocols, thus, taking full advantage of the capabilities of the Bluetooth technology. Even though great effort was made by the SIG to keep the complexity of the Bluetooth architecture to a minimum, it certainly cannot fit inside a single chapter. Therefore, most of the attention will be drawn upon the functionality, the role, the special features, the interface and the overall rationale of every layer of the Bluetooth stack. The layers especially examined are the Bluetooth-specific protocols of the transport and MW groups (which are also utilized in this thesis) whereas a brief description of the rest protocols -applications group- will follow. Before going on with the details, it should be noted that a brief overview of all the protocol layers and technical characteristics is included in Appendix A (fig. A.1).

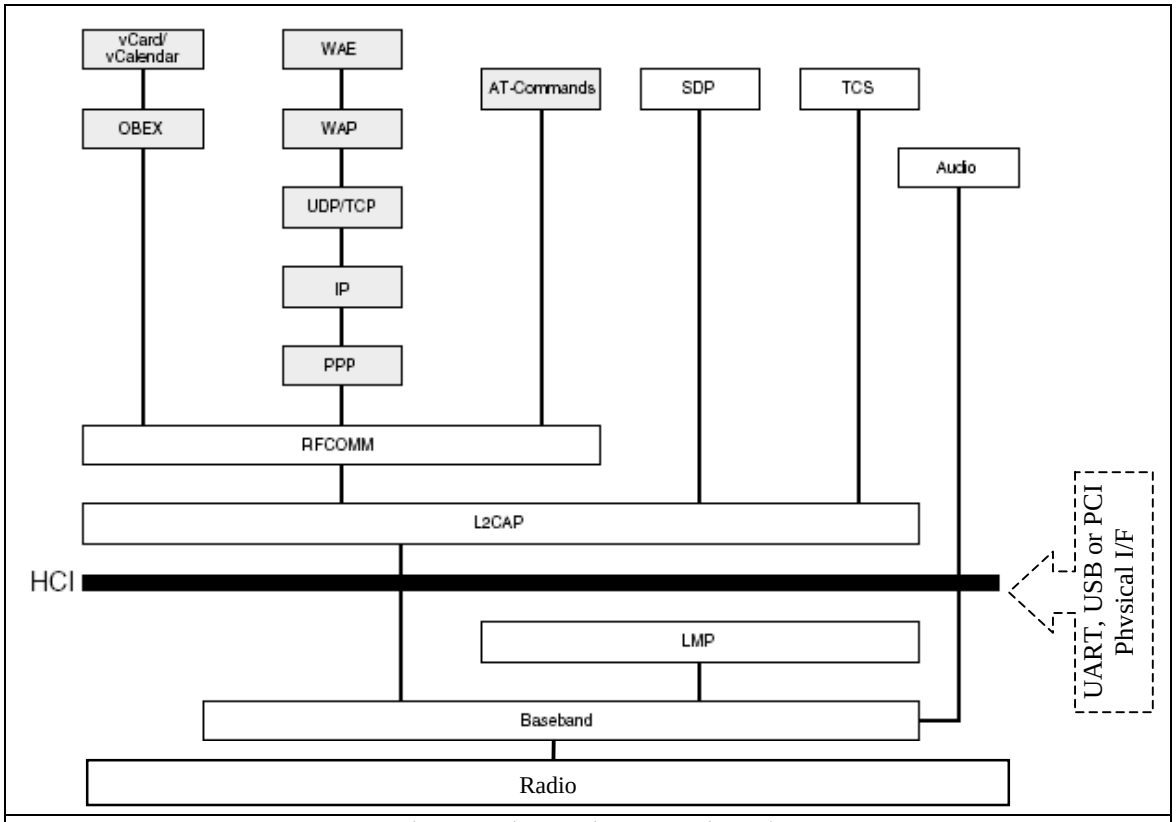


Fig. 3.1: Bluetooth Protocol Stack

3.2.1 Radio (RF)

The Bluetooth system operates in the 2.4 GHz ISM band. The ISM bands are license-free bands set aside for use by industrial, scientific and medical wireless equipment; nonetheless, these bands are under few but strict regulations (spread-spectrum pattern, low power). The 2.4 GHz ISM band range and structure is shown in figure 3.2:

2.4 GHz ISM band (MHz)	**Frequency channels (MHz) k = 0, 1, 2, …, m-1**	**Lower Guard Band (MHz)**	**Upper Guard Band (MHz)**
2,400.0 – 2,483.5	2,402 + k, with m = 79	2.0	3.5
Fig. 3.2: Frequency allocation in the 2.4 GHz band			

The Bluetooth transceiver is a **frequency-hopping spread-spectrum** (FHSS) radio system operating over a number m of 1 MHz-wide channels (as shown in fig. 3.2). While for the majority of countries m = 79, regulations in certain countries (Japan, Spain and France) constrain the 2.4 GHz band to a narrower one with m = 23 although this is due to change in the near future. The FHSS pattern is very suitable for avoiding interference in this heavily-crowded and -therefore- noisy band. It is well suited for low-power, low-cost radio implementations and is used in many WLAN products (e.g. microwave ovens, garage-door openers, cordless phones etc.). The Bluetooth specification defines a high hop rate of 1600 hops per second instead of just a few hops per second used in other implementations (e.g. 2.5 hops per second for the 802.11 protocol). Every frequency channel is used for 625 μs (one time-slot) followed by a hop, in a pseudo-random order, to another channel for another 625 μs transmission, repeated constantly. That way, the Bluetooth traffic is spread over the entire ISM band and a very good interference-protection scheme is achieved. If one of the transmissions is jammed, the probability of interference on the next hop channel is very low. Furthermore, error correction algorithms are used to correct the fault caused by jammed transmissions. The use of direct-sequence spread-spectrum (DSSS) systems, which are also permitted in the 2.4 GHz ISM band, might be prohibitively costly for the low-cost requirement of Bluetooth radios.

The BT spec. defines a receiver sensitivity of -70 dB. The radio transmitting power is typically defined to 1 mW (0 dBm, Class 3 device) but also a 100 mW antenna scheme is included in the BT spec (20 dBm, Class 1 device). The low-power consumption implies that a Bluetooth unit can operate on the power

from a small battery for a long time. These HW characteristics make it possible to fit a Bluetooth unit in many electrical devices. The maximum Bluetooth range is 10 m (Pout = 1 mW), with a possibility to extend it to 100 m (Pout = 10 mW). The maximum symbol rate is 1 Msymbol per second; using the binary GFSK, this translates into a 1 Mbps raw link speed (i.e. T_{bit} = 1 μsec). However, the maximum effective payload is lower because of the overhead of the different protocol layers over the radio layer. Estimates have indicated a maximum over-the-air data transfer rate (asymmetric) of 723.2 kbps (transmission) and 57.6 kbps (reception) or 433.9 kbps for a symmetric one. For full duplex transmission, a **Time-Division Duplex** (TDD) scheme is used (examined below).

3.2.2 Baseband (BB)

While the radio deals with the acts of sending and receiving data over the air, the Baseband and Link Control layer enable the physical RF link between Bluetooth units. Its functions are vast and diverse including device synchronization, connection establishment (paging & inquiry), frequency-hop selection, support for all link types (ACL, SCO), connection power modes (active, sniff, hold, park), security algorithms etc..

3.2.2.1 Master – Slave roles

At the baseband level, when two devices establish a link, one acts in the role of the **master** and the other in the role of the **slave**. The role of the master does not imply special privileges; instead, it is the master system clock and the master identity that are the central parts in the frequency-hopping pattern (examined below). The next hop channel is determined by the hop sequence and by the phase in this sequence. The identity of the master determines the sequence and the master system clock determines the phase. All slaves communicating with a given master hop together in unison with the master. The master role is -generally speaking- assumed by the device that initiates the communication. A given master may communicate with multiple slaves –up to 7 active slaves and up to 255 parked slaves (the terms "active" and "parked" are described below). All slaves communicating with a single master form a **piconet** (the term used by the BT spec. for a PAN); only one master can be present in a single piconet. This master-slave distinction -in general- is of no importance to the higher layers of the Bluetooth stack and often is transparent to the application layer.

3.2.2.2 Connection power modes

As noted above, a piconet can include up to 7 active slaves and 255 parked slaves[13]. Apart from the "**active**" mode, the "**park**" mode along with "**sniff**" and "**hold**" are the three possible connection modes defined by the BT spec. All of them are connection modes that adjust the performance, power and number of attached devices to a piconet. When no connection is running, the baseband is said to be in a "**standby**" mode. In active mode a slave must essentially listen for all transmissions from the master. Active slaves receive packets that enable them to remain synchronized with the master and that inform them when they can transmit packets back to the master. The active state typically provides the fastest response time but also typically consumes the most power since it is always receiving packets and is always ready to transmit packets. Sniff mode is a way of reducing power consumption. In this mode a slave essentially becomes active periodically. This is achieved through an "agreement" between the master and the specific slave; the master transmits packets destined for this slave only at certain regular intervals. Likewise, the slave needs only listen for packets from the master only at the start of such an interval. If no packet is present, then the slave can sleep till the next interval. Obviously, sniff mode provides a more effective power management but less responsiveness, both depending on the length of the sniff interval. In hold mode a slave may stop listening for (certain or all types of) packets for a specified time interval. The master and slave agree upon a hold time and the communication link is "quiet" for that amount of time. Although hold mode can be even more power-saving and less responsive that sniff mode, this also depends largely on the hold time interval and on the slave actions during that interval. Finally, in park mode, a slave maintains synchronization with the master (by listening to the master periodically) but is no longer considered active (as opposed to the previous cases). The reason is that when a slave joins a piconet, it is assigned a unique 3-bit address (**AM_ADDR**, active mode address); that is why only 7 slaves can be connected simultaneously to a master (given the AM_ADDR value "0b000" is reserved). In sniff or hold mode a slave is considered a fully qualified member of the piconet and -as such- it maintains its AM_ADDR. On the contrary, when in park mode -in order to further reduce power consumption- a slave gives up its AM_ADDR and acquires a unique 8-bit address (**PM_ADDR**, park mode address)[14]. That is the reason for which 255

[13] In fact, more than 255 parked slaves are possible. The BT spec. defines "direct" addressing for up to 255 parked slaves via the PM_ADDR (see below) and, also, "indirect" addressing of parked slaves via their specific Bluetooth device address or BD_ADDR (also seen below).

[14] Actually, an additional 8-bit address is assigned, the AR_ADDR (access request address) which is used to schedule the order of readmission of parked slaves to the piconet in a way that minimizes the possibility of collisions. For more details see [3]: Part B.

parked slaves can be supported in a piconet (given the PM_ADDR value "0x00" is reserved). Even though synchronization is preserved among the parked slaves and the master, the time penalty for reentering the piconet (reassignment of an AM_ADDR etc.) is the price to pay for the reduced power consumption. However, the "park mode" feature expands the piconet capacity from 8 to 256 coexisting members, which are more than sufficient in most PAN cases.

3.2.2.3 Bluetooth network topology

Taking a broader look at the Bluetooth network model -given the previous discussion of the master/slave roles and baseband modes- it can be argued that it is one of peer-to-peer communications based upon proximity networking (also known as "**ad hoc**" networking). This means that when two devices come within range of each other (as specified in the "Radio" section), they can automatically establish a communication link. The coexistence of a master device and one or more slave devices constitutes the -above seen- piconet. All of the devices in the piconet are synchronized, hopping together. In the vicinity of the piconet can also be more devices. They may as well be in standby mode or to be connected to this or another master (from a different piconet). When two or more piconets at least partially overlap in *time* and *space*, a **scatternet** is formed. Any device can be part of more than one piconet; it can even be a master in one piconet and a slave in another. This scatternet topology is an extremely *flexible* method for the devices to maintain multiple connections. An example of the piconet/scatternet topology is depicted in figure 3.3.

3.2.2.4 Bluetooth Device Address – Bluetooth Clock

One major issue concerning the baseband protocol and, at the same time, the key functionality that hides behind all Bluetooth communication is the issue of connection establishment. Before delving into its details, though, two fundamental elements of any Bluetooth device are examined. The **Bluetooth Device Address** (**BD_ADDR**, for short) is a unique static 48-bit address "hardwired" to every Bluetooth device. This BD_ADDR is an IEEE-standardized address similar to the MAC[15] address of IEEE 802.XX LAN devices. It is partitioned into three parts: the 24-bit *lower address part* (LAP), the 8-bit *upper address part* (UAP) and the 16-bit *non-significant address part* (NAP)[16]. The various parts of the BD_ADDR are involved in nearly every operation of the baseband. The second key element of any

[15] MAC: Medium Access Control

[16] The UAP and NAP constitute the Organization Unique Identifier (OUI) and are uniquely assigned to every Bluetooth product whereas the LAP is assigned internally by various organizations.

Bluetooth device is the **Bluetooth Clock**. It is a free-running 28-bit native clock that is never adjusted and never turned off. The clock ticks 3200 times per second or once every 312.5 μsec i.e. has a frequency of 3.2 KHz, which is twice as fast as the frequency-hopping scheme (for reasons explained later on). It is the clock of the master that defines the value and phase of the frequency-hopping pattern of a piconet. Since the Bluetooth clock cannot be altered, all the slaves entering a piconet manage to get synchronized by adding an offset to their own (native) clock in such a manner that: $CLK_{master} = CLK_{slave} + CLK\_OFFSET$. In this way, all Bluetooth devices in a piconet have a unified clock and manage to hop together. Naturally, the clock of a slave may present some skew from that of the master as time elapses, so re-sync is required. This task is achieved by the master periodically transmitting some types of baseband packets that carry the current value of the master clock to the slaves which are, then, able to recalculate their clock offsets.

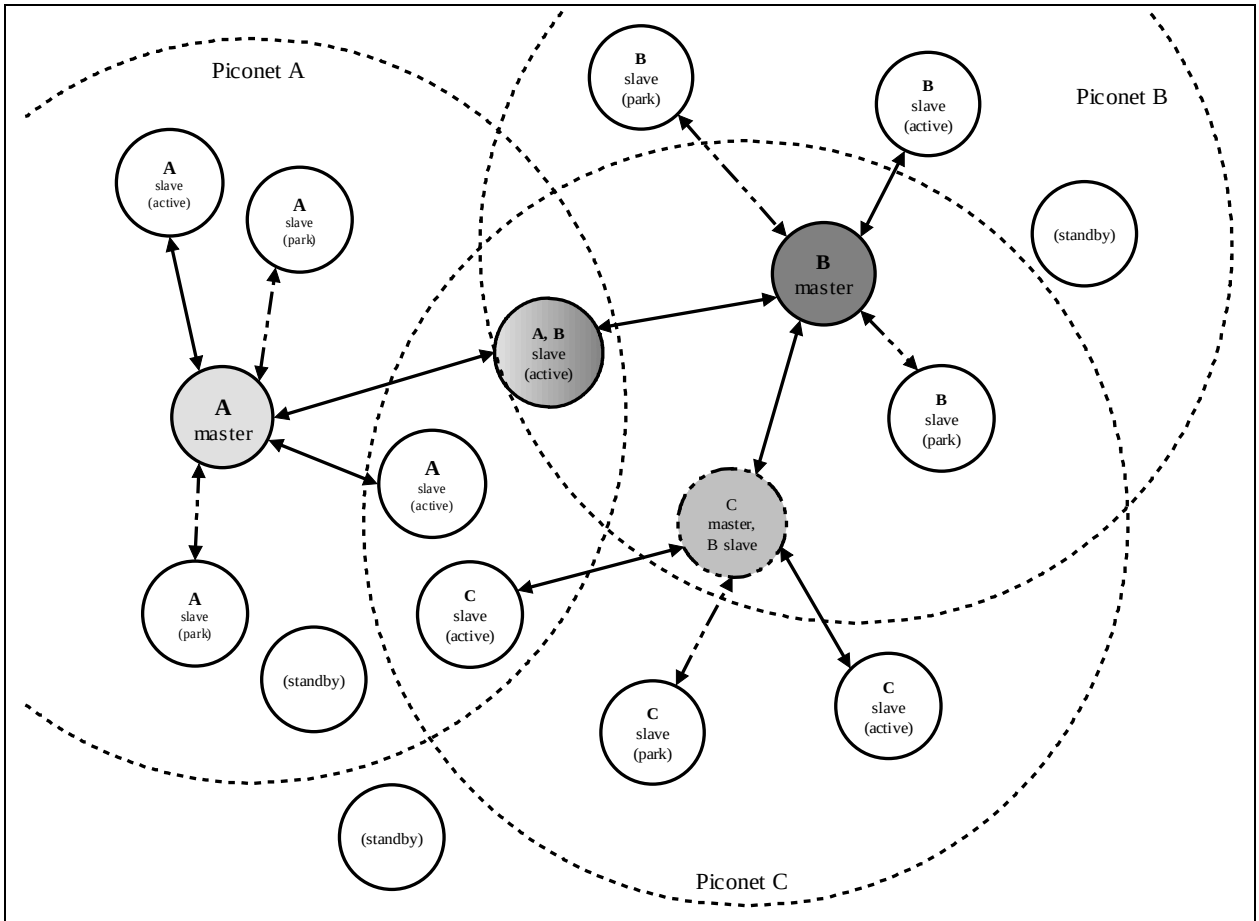


Fig. 3.3: Bluetooth piconet & scatternet topology

3.2.2.5 Connection details

Till now the formation of a piconet was taken for "granted" since no procedures of setup and connection were addressed. To begin with, let's take a look at figure 3.4 which displays the various operational states a Bluetooth device can undergo and the connection among them.

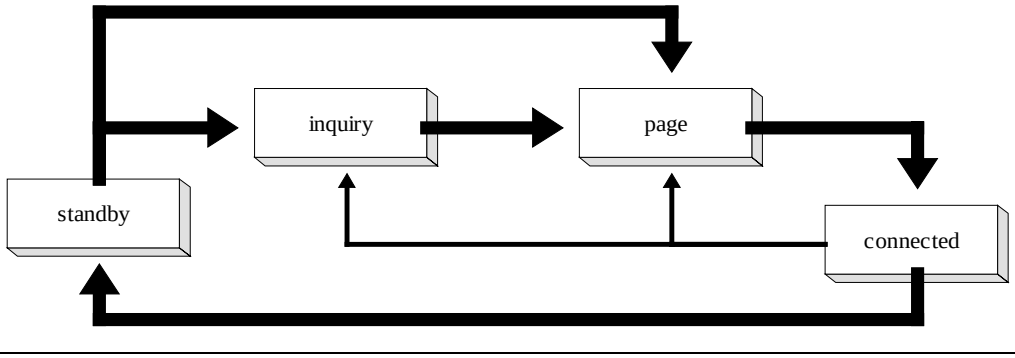


Fig. 3.4: Operational states of a Bluetooth device

The standby state is the default (low-power) operational state for a Bluetooth device. On moving to a connected state, a device goes through the **inquiry** and **page** states. In the *inquiry* state, a device "searches" for other devices in its vicinity. These other devices must be in an *inquiry* scan state to listen and respond to inquiries. In a similar manner, a device in the *page* state explicitly "invites" another device to join the piconet whose master is the inviting device. The other device must be in the *page scan* state to listen and respond to pages. As figure 3.4 shows, a device may bypass the inquiry state if the identity of the device to be paged is already known.

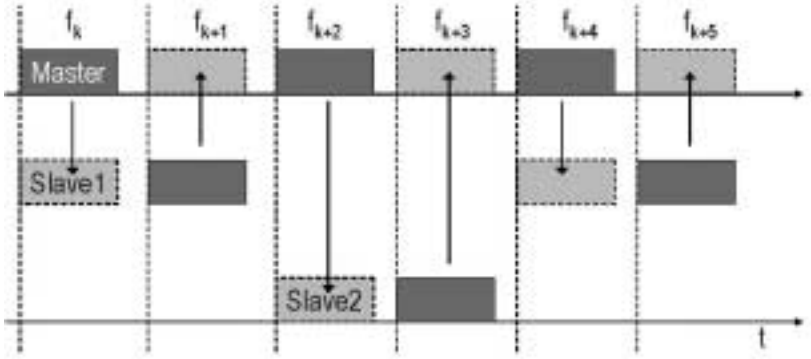
Every 312.5 μsec (LSB of Bluetooth clock) an inquiring device selects a new frequency from the 79-channel 2.4 GHz ISM band at which to transmit an inquiry to an inquired device[17]. On the other side, during inquiry scans, a device listens for transmitted inquiries changing its "listening" frequency every 1.28 seconds. Obviously, the two devices are not yet synchronized so the clock used by the inquired (slave) device is only a good estimate of the inquiring (master) device, based upon their most recent communication, if any. This is the main reason why the inquiring device changes frequencies at a much higher rate than the inquired device; the master is much (4000 times) faster than the slave so as to discover it before changing its frequency. Yet, when two devices manage two "see" each other through

[17] That is why the Bluetooth clock has a finer granularity of 312.5 μsec than the one needed for simple data transmission: 625 μsec.

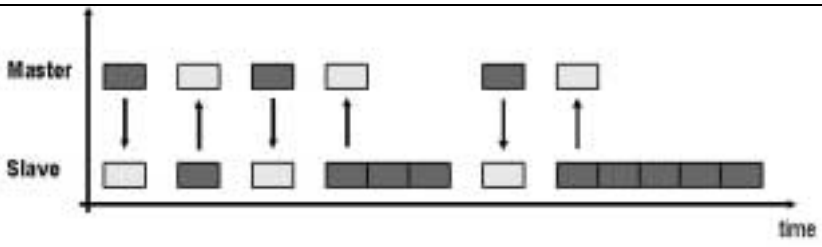
an inquiry, data about the master clock are sent over to the inquired device. The pattern according to which inquiries are performed is a very specific one called *inquiry-hopping sequence* and is a set of frequencies from which both the inquiring and the inquired devices draw their next frequency. Just as in an inquiry operation, a paging device selects a new frequency at which to transmit a page every 312.5 μsec whereas paged devices executing page scans, select a new listening frequency every 1.28 seconds. The pattern used in this case is called *page-hopping sequence*. Each of these sequences is a well-defined, periodic sequence composed of 79 frequencies uniformly distributed over the 79 frequency channels of the 2.4 GHz ISM band. The period of each sequence is 32 hops.

Through inquiries and pages connections are built and normal piconet operation is established. In this state, the pattern used is called *channel-hopping sequence* and it is the one briefly explained in the "Radio" section; a new frequency from the channel-hopping sequence is selected every 625 μsec. This residence time at a specific frequency is called *slot*; during a slot a device may transmit a single packet of information hereafter referred to as a **baseband packet data unit** (**BB_PDU**). However, the residence time at a frequency may occupy multiple slots (with a maximum of 5), thus permitting multi-slot BB_PDU transmissions to occur. Following a multi-slot transmission, the next frequency selected is the one that would have been used if a single-slot transmission had occurred instead. With a common clock reference among all devices in a piconet (master's clock), the transmission time on the piconet is divided into master and slave transmission slots. A master starts its transmissions on even- while the slaves on odd- numbered slots, exclusively. A particular slave transmits if and only if the last master transmission was destined exclusively to this slave. Thus, the medium access protocol for Bluetooth communications is a packet-based *time-division-duplex* (TDD)[18] polling scheme- as previously seen. As mentioned earlier, multi-slot transmissions are possible. However, due to this TDD scheme, multi-slot transmissions are limited to an odd number of slots (1, 3 or 5) in order to guarantee that master transmissions always start on even slots and slave transmissions on odd slots. Figure 3.5 presents one example of a transmission to multiple slaves and one of multi-slot transmission.

[18] TDD refers to a time-division multiplexing technique where the transmission time on a single communication channel is divided into successive non-overlapping intervals, every each other of which is used for transmissions in one of two opposing directions. The transmission direction alternates with each successive interval.



Multi-slave transmission



Multi-slot transmission

Fig. 3.5: Data transmission patterns

3.2.2.6 BB_PDU general format

A brief insight on the BB_PDU structure is given here; its contents are depicted in figure 3.6. First of all, it should be noted that all Bluetooth transmissions start with the LSB and proceed with the MSB (i.e. *Little-Endian* transmission order). The **access code** is used as a piconet identifier and assumes different values depending on whether the specific BB_PDU is aimed for inquiring, paging or channel (mere data exchange) purposes[19].

Access Code	**Header**	**Payload**
- 68 | 72 bits -	- 54 bits -	- 0 to 2745 bits -

… (x3) …

AM_ADDR	**TYPE**	**FLOW**	**ARQN**	**SEQN**	**HEC**
- 3 bits -	- 4 bits -	- 1 bit -	- 1 bit -	- 1 bit -	- 8 bits -

Fig. 3.6: BB_PDU format

[19] The access codes used are: GIAC or DIAC (general or dedicated inquiry access code), DAC (device access code) and CAC (channel access code), respectively. For more details on the various types of access codes and their use see [3]: Part B.

The next field, the **header**, is divided in the following subfields:

- AM_ADDR: active member address of a device being active in a piconet (as examined previously). Note that AM_ADDR value: "0b000" is used for *broadcasting* packets to all active slaves of a piconet (as opposed to unicasting to specific slaves)
- TYPE: defines 16 BB_PDU payload types
- FLOW: stop/go flow control switch set by a receiving device in its response to the sender
- ARQN: used for acknowledging successfully transmitted BB_PDUs (default value: NAK)
- SEQN: simple (odd/even) sequence number for filtering out duplicate transmissions
- HEC: *header-error-check* generated by polynomial: $G_{HEC}(x) = x^8 + x^7 + x^5 + x^2 + x + 1$

The previous data are used to help medium access control and add up to 18 bits but since a 1/3 *forward-error-correcting* (**FEC**) code is used, every bit of the header is transmitted 3 times in sequence, for reliability.

3.2.2.7 Link types: ACL, SCO

The Bluetooth baseband supports two link types over a piconet. **Synchronous Connection-Oriented Links** (**SCO**) are symmetric, point-to-point links between the master and a single slave. An SCO link reserves slots a priori and can therefore be considered as a *circuit-switched* connection between the master and the slave. The SCO link typically supports time-bounded information like voice (effectively providing a bit rate of 64 kbps). Even the length of the baseband slot, 625 μsec, is selected to minimize the D_{pack}[20] for audio traffic and the effects of noise interference. The master can support up to three SCO links to the same slave or to different slaves. A slave can support up to three SCO links from the same master or two SCO links if the links originate from different masters. SCO packets are never retransmitted by the source and never acknowledged by the destination. The master will send SCO packets at regular intervals. The SCO slave is always allowed to respond with an SCO packet in the following slave-to-master slot unless a different slave was addressed in the previous master-to-slave slot. If the SCO slave fails to decode the slave address in the packet header, it is still allowed to return an SCO packet in the reserved SCO slot. The SCO link is established by the master sending an SCO setup message via the LMP. **Asynchronous Connection-Less Links** (**ACL**) are point-to-multipoint links between the master and all the slaves participating on the piconet. In the slots not reserved for the SCO

[20] D_{pack}: Packetization delay

link(s), the master can establish an ACL link on a per-slot basis to any slave. The ACL link provides a *packet-switched* connection between the master and all active slaves participating in the piconet. Both asynchronous and isochronous services are supported. Between a master and a slave only a single ACL link can exist. For most ACL packets, packet retransmission is applied to assure data integrity. A slave is permitted to return an ACL packet in the slave-to-master slot if and only if it has been addressed in the preceding master-to-slave slot. If the slave fails to decode the slave address in the packet header, it is not allowed to transmit. ACL packets not addressed to a specific slave are considered as broadcast packets and are read by every slave. Note that prior to establishing an SCO link, an ACL link must already exist to carry, at a minimum, the SCO connection control information. Note, also, that ACL and SCO packets reside inside the BB_PDU payload and have their own inner structure. In a nutshell, SCO links support *real-time voice traffic* using reserved bandwidth. ACL links support *best-effort traffic.*

There are 3 packet types for SCO links: HV1, HV2, and HV3. HV stands for High-quality Voice. These all transmit in a single slot frame but feature a range of trade-offs in capacity, bandwidth required, and error susceptibility as follows:

- HV1 packets contain only 1.25 msec of audio data, but this data is well protected from errors. Even in cases where recovery still proves impossible, the receiver at least knows there was an error and can drop this data from the audio stream creating a silent segment of noise. The downside of HV1 links is that they require the ENTIRE Bluetooth transmission capacity.
- HV2 packets contain 2.5 msec of audio and protect it with moderate FEC encoding i.e. can recover data in fewer error instances than with HV1. In exchange for this reduction in error protection HV2 links only require 1/2 the bandwidth of HV1 or 1/2 of Bluetooth's transmission capacity.
- HV3 packets contain 3.75 msec of audio with no error detection or recovery. Thus HV3 links are subject to errors and can induce noise into the audio stream. Their advantage is that they only require 1/3 of Bluetooth's transmission capacity.

ACL links have 6 options: D(M|H)(1|3|5) but they are divided into 2 basic categories: DM (Medium-rate Data) and DH (High-rate Data). One option trades off FEC for payload capacity (DM vs. DH); so DM packets have FEC and can avoid retransmission in the case of many errors but DH packets still have CRC so errors will be detected and retransmission will ensure (eventual) correct delivery. The second choice simply adjusts the bandwidth allocated to the packet by selecting among 1/1, 3/1, and 5/1 multi-slot frames. In 1/1 frames the outgoing transmission packet must fit into a single 625-µsec

slot. In 3/1 and 5/1 multi-slot frames this is extended to an 1875-μsec triple and 3125-μsec penta slot respectively. Multi-slot frames increase payload capacity very efficiently for two reasons. Firstly, they pay only a single header overhead cost, so the header space in the 2nd and on slots can be used by payload. Second, they lose less of their transmission window to Bluetooth timing margins. In a 625-μsec slot some tens of microseconds are reserved for the timing margin around the packet. 3/1 and 5/1 frames have the same requirement, but this overhead drops proportionately to the increase in the transmission window.

In addition to the above packet types, there is a hybrid packet called DV (Data – Voice) that combines both SCO and ACL data. On the SCO part, it carries 1.25 msec of voice with no FEC (i.e. HV1 capacity with HV3 strength) whereas, on the ACL part, it carries up to 9 data bytes with FEC and CRC (i.e. approximately 1/2 of DM1 capacity). In figure 3.7 the Bluetooth link types are summarized. It can be argued that Bluetooth has a great deal of flexibility designed to maximize the utility of its links in a wide range of environments. In low density / low noise environments Bluetooth can utilize high capacity and low overhead links to maximize bandwidth. In high density/high noise environments it can fall back to more conservative and robust links, albeit with some increased overhead, to maintain reliable operation. In Appendix A, a graphical addendum to the various ACL & SCO link types is given (fig. A.2, A.3, A4.).

Packet Type	Payload Header (bytes)	User Payload (bytes)	FEC	CRC	Maximum Symmetric Data Rate	Maximum Asymmetric Data Rate	
HV1	N/A	10	1/3	No	64.0Kbps		
HV2	N/A	20	2/3	No	64.0Kbps		
HV3	N/A	30	No	No	64.0Kbps	Forward	Reverse
DV (voice part)	N/A	10	No	No	64.0Kbps	N/A	N/A
DV (data part)	1	0-9	2/3	Yes	57.6Kbps	57.6Kbps	57.6Kbps
DM1	1	0-17	2/3	Yes	108.8Kbps	108.8Kbps	108.8Kbps
DH1	1	0-27	No	Yes	172.8Kbps	172.8Kbps	172.8Kbps
DM3	2	0-131	2/3	Yes	258.1Kbps	387.2Kbps	54.4Kbps
DH3	2	0-183	No	Yes	390.4Kbps	585.6Kbps	86.4Kbps
DM5	2	0-223	2/3	Yes	286.7Kbps	477.8Kbps	36.3Kbps
DH5	2	0-339	No	Yes	433.9Kbps	723.2Kbps	57.6Kbps

Fig. 3.7: ACL & SCO link characteristics summary

Finally, the format of a typical ACL packet (DM packet or ACL portion of DV packet) is presented in figure 3.8.

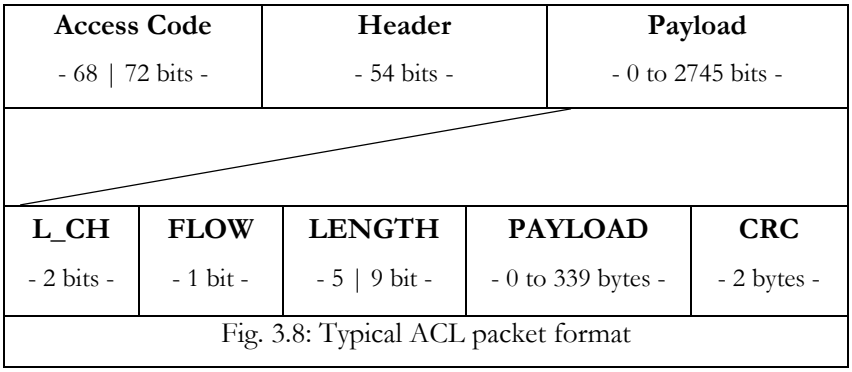


Fig. 3.8: Typical ACL packet format

These subfields reside inside the payload of a BB_PDU packet and concisely are:

- L_CH (logical channel): '0b01': continuation segment of (upper) L2CAP packet (examined later on), '0b10': start of L2CAP packet, '0b11' LMP packet (examined next), and '0b00' reserved for future use.
- FLOW: flow control on the ACL link
- LENGTH: length of ACL payload (excludes header & CRC) for single- and multi-slot packets
- PAYLOAD: ACL packet payload spanning 1 to 5 slots
- CRC: *Cyclic-Redundancy-Check* for payload protection; generator polynomial: $G_{CRC}(x) = x^{16} + x^{12} + x^5 + 1$

3.2.2.8 Bit-stream processing

Apart from all the above, the baseband is also responsible for all bit-stream processing immediately before and after RF transmission. A typical processing dataflow is depicted in figure 3.8.

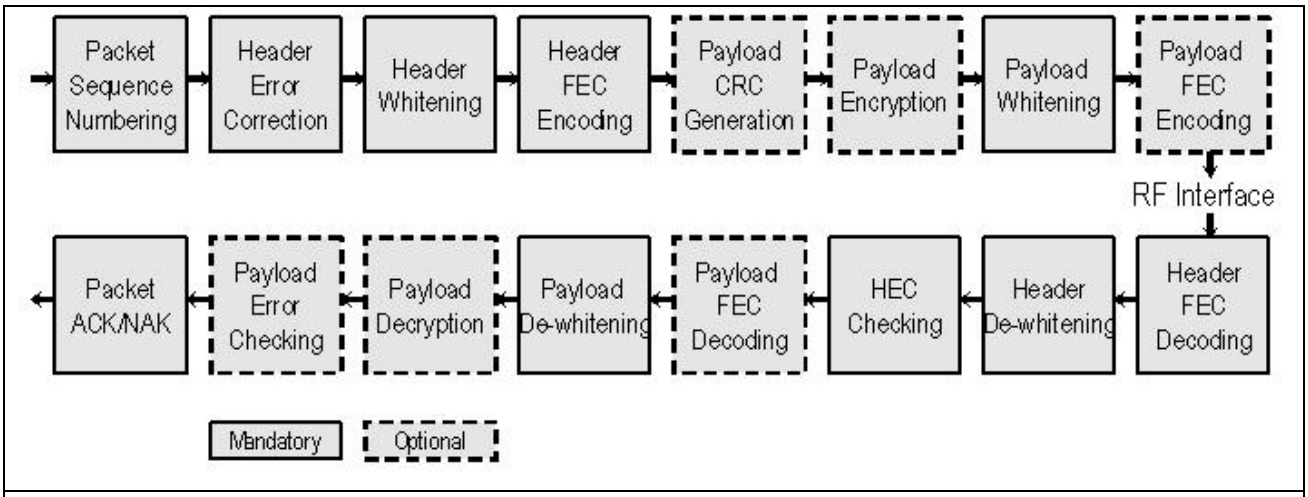


Fig. 3.8: Baseband bit-stream processing

3.2.3 Link Manager Protocol (LMP)

While the baseband takes care of a plethora of low-level Bluetooth issues and is a key layer to the overall Bluetooth functionality, the Link Manager undertakes the more "boring" task of negotiating the properties of the Bluetooth air interface between communicating devices; it performs all link creation, management, and termination operations. LMP messages are used for link setup, security and control and are not propagated to the upper layers of the stack. Such LMP messages are hereafter noted as LMP_PDUs. They only carry control data and no application data. The LMP_PDUs, belonging to a layer superjacent to the baseband, are carried in the payload field of ACL data packets (with the "L_CH" field marked as '0b11'). The LMP_PDU contents are shown in figure 3.9.

transactionID	**OpCode**	**payload**
- 1 bit -	- 7 bits -	- 0 to 17 bytes -

Fig. 3.9: LMP_PDU format

Their functionality is the following:

- transactionID: '0b0': identifies a LM transaction initiated by the master, '0b1': identifies a LM transaction initiated by a slave
- OpCode: identifies the LMP_PDU and type of contents it carries
- payload: the payload of a LMP_PDU

Link managers among devices communicate using a challenge-response approach i.e. one link manager issues a transaction request and another one must respond either negatively or positively. The issues settled are security, QoS[21] and power management.

3.2.3.1 Security

Security comes in two flavors: **device authentication** and **link encryption**. The former is mandatory and is enabled through the use of specific key, either static or dynamic ones. These keys are link keys only known to the communicating devices ensuring that these alone will establish a link among them else a connection fails. Of course, there are cases when a device is open to any device wanting to connect with it (e.g. that would be the case of a wireless access point), in which case a public or shared key is used.

The latter security measure is link encryption and is optional, always following device authentication. It is based on a 1-bit stream cipher (included in the BT spec.). The size of the encryption, which changes with every new BB_PDU transmission, is negotiable to match application requirements. The encryption key is derived by the link key used during the authentication of devices. The maximum key size is 128 bits and is applied only to the payload of the BB_PDU (both ACL and SCO links).

3.2.3.2 Other LMP tasks

Power management based on the various connection power modes -active, sniff, hold, park- discussed in the previous section ("Baseband") can be initiated and terminated in this layer. Also, bandwidth management can be achieved through the LMP. SCO packets are prioritized over all ACL traffic (but not over control traffic) to provide high-quality voice transmission. Bandwidth guarantees can be drawn for ACL links also through polling interval restrictions for the slaves. This means that control can be applied over the minimum bandwidth assignment for ACL traffic or -equivalently- over the maximum access delay of ACL BB_PDUs. While a master or slave can request a change in the QoS of a specific ACL link, which is negotiated with sequential LMP_PDUs among them, a master can enforce a new QoS scheme this giving the master a clear and straight way of controlling the physical medium. This feature of the link manager is available for the SCO links also but in this case it is optional -as opposed to the ACL links.

[21] QoS: Quality of Service

Finally, the link manager performs some tasks related to the link controller and the baseband protocol such as negotiation of the used paging scheme[22], link key change, master-slave role switch etc.. This switch is occasionally necessitated by some higher-level applications, e.g. in the case of building a wireless LAN using PPP. During the switch, crucial information of the master is transferred to the slave becoming the new master of the piconet. The whole "package" of tasks to be executed is a responsibility of the LMP. Also, link managers typically exchange information about each other to better coordinate their interactions, e.g. *supported LMP version*, *supported features* (including the optional features supported by the Radio, BB & LM), *name* (including the user-friendly name of the remote device) etc.. Last but not least, the link manager is responsible for sending the attachment/detachment signals for a requested connection and also negotiates the parameters of the link.

3.2.4 Logical Link Control & Adaptation Protocol (L2CAP)

The primary role of the L2CAP is to hide the peculiarities of the lower-layer transport protocols from the upper layers. In so doing, a large number of already developed higher-layer protocols and applications can be made to run over Bluetooth links with little, if any, modification. The support of such protocols is the key element to the so-called Bluetooth interoperability.

The L2CAP concerns itself only with asynchronous information (ACL packets), as it can be seen from the Bluetooth protocol stack (fig. 3.1). Its packets, hereafter referred to as **L2CAP_PDU**s, are carried on BB_PDUs whose L_CH field in the payload header has the value '0b10' or '0b01' (as seen in the "Baseband" section). This protocol's purpose is threefold:

1. **higher-level protocol multiplexing**, compensating for the lack of support at the lower transport layers (since no "type" field identifying the various higher layer protocols has been declared so far),
2. **packet segmentation and reassembly** (**SAR**), for building larger, higher-layer packets (for the more demanding higher-layer applications) from the smaller baseband packets, and
3. **conveying of QoS information**, which aids in controlling the transmission resources in a way that supports the expected QoS.

The L2CAP layer assumes reliable transmission of its PDUs from the underlying layers and permits higher level protocols and applications to transmit and receive L2CAP data packets up to 64 KB in length. Communication between L2CAP layers is based on *logical links* called **channels**, through which

[22] See [3]: Part C, for more details.

L2CAP traffic flows between endpoints within each device. Each endpoint of a channel is assigned a unique **channel identifier** (**CID**), a 16-bit number which is strictly *locally* administered. Depending on the CID value, various types of channels can be formed, as identified in figure 3.10.

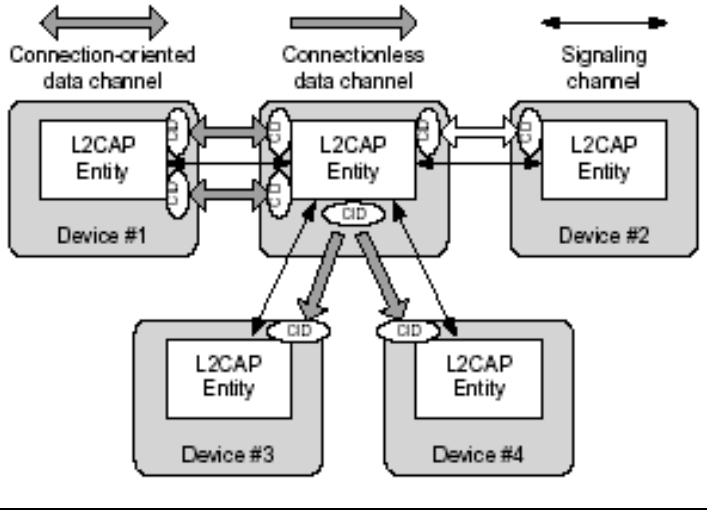


Fig. 3.10: Channels between devices [3]

There are persistent *connection-oriented* (CO) channels used for bidirectional communications; also, ephemeral *connectionless* (CL) channels that are unidirectional and can be used for broadcast transmissions to groups of devices. Finally, there are *signaling* channels that are used primarily to exchange control information that is used to establish and configure CO channels. The CID values assigned locally for each of the above types of channels are well-defined and are shown in figure 3.11.

CID	Details
0x0000	Null identifier; not to be assigned
0x0001	CID for both endpoints of an L2CAP signaling channel
0x0002	CID for the destination endpoint of a CL L2CAP channel
0x0003 – 0x003F	Reserved
0x0040 – 0xFFFF	Dynamically allocated CIDs (on demand by a device to its local endpoints for CL and CO L2CAP channels)

Fig. 3.11: CID types

There are two types of L2CAP_PDUs: the first is used with CO channels and the second with CL channels. Signaling L2CAP_PDUs are formed according to the former type. In short the CL and CO packet formats are depicted in figures 3.12 and 3.13, respectively.

Header			Payload
Length	**Destination_CID**	**PSM**	**Payload**
- 2 bytes -	- 2 bytes -	- ≥ 2 bytes -	- (Length - PSM) bytes -
Fig. 3.12: L2CAP_PDU format (CL)			

Header		Payload
Length	**Destination_CID**	**Payload**
- 2 bytes -	- 2 bytes -	- (Length) bytes -
Fig. 3.13: L2CAP_PDU format (CO)		

The various fields are the following:

- Length: total length in bytes of a CL L2CAP_PDU excluding the Length and CID fields
- Destination_CID: indicates the CID of the destination endpoint of the L2CAP channel used for transmission (equal to '0x0002' for CL-destined packets)
- PSM: protocol and service multiplexer (identifies the destination payload processing entity for the transmitted L2CAP_PDU, e.g. a higher-layer application)
- Payload: payload data with maximum size 64 KB (minus the 2 bytes of the PSM filed, if any)

CO L2CAP channels use signaling to become established, configured and terminated. L2CAP signaling is also based on request-response transactions. During connection establishment, channel properties are negotiated through such transactions, effectively defining the QoS. Further settings can be configured during the connection via the signaling channel controlling the specific CO channel[23].

3.2.5 Host Controller Interface (HCI)

Special attention will be garnered in the HCI because it's the layer that this thesis is specifically related with. The application suite built, lies on top of this layer and -ergo- getting to know its functionality and terminology is of utmost importance.

The Bluetooth transport protocols can be implemented in an integrated fashion entirely on the same host (e.g. motherboard or processor) that runs the applications that make use of those protocols. On the other hand, they may be implemented independently of the host on a separate Bluetooth module that is -then- attached to the host as an add-on accessory attachment or a plug-in card, through some

[23] See [3]: Part D, for more details.

physical interface on the host such as USB, RS-232, UART or PCI interface (these are the 1-CPU and 2-CPU architectural choices introduced in chapter 2). When implemented separately, the module also contains a **host controller** unit (**HC**) whose responsibility is to interpret the information received by the host and direct it to the appropriate components of the module, like the link manager or the link controller. Likewise, the HC collects data and HW/FW status from the module and passes it to the host as needed. As discussed in chapter 2, a typical implementation places the radio, link manager, link controller and physical interfaces to the host inside the module, without precluding the possibility of another partition of the Bluetooth protocol stack. For maximum-interoperability purposes, the SIG has defined standardized physical I/Fs[24] (RS-232, USB etc.). The SIG has also defined a transaction-style communication protocol to carry information between the host and the HC. The physical interface along with this communication protocol is collectively referred to as the *host controller interface* (HCI). A protocol stack with an HCI appears in figure 3.14 also showing the implementation nature of every protocol. As presented, the HCI contains an *HCI driver* which executes the communication protocol for exchanging HCI traffic with the HC and a *physical transport driver* which is the SW underlying the operation of the selected HW interface (USB, RS-232 etc.). Since the HCI is merely a means of porting a host with a Bluetooth module and may as well be omitted, it ought to be absolutely *transparent* to its subjacent and superjacent layers, i.e. it should not offer any added functionality or service to the Bluetooth operation (like the LMP or L2CAP do), which is true. Strictly speaking, the HCI is not a communications protocol. However, the HCI specification defines formats for packets that cross a host interface and associations between these packets. These formats and associations are key elements of a protocol specification.

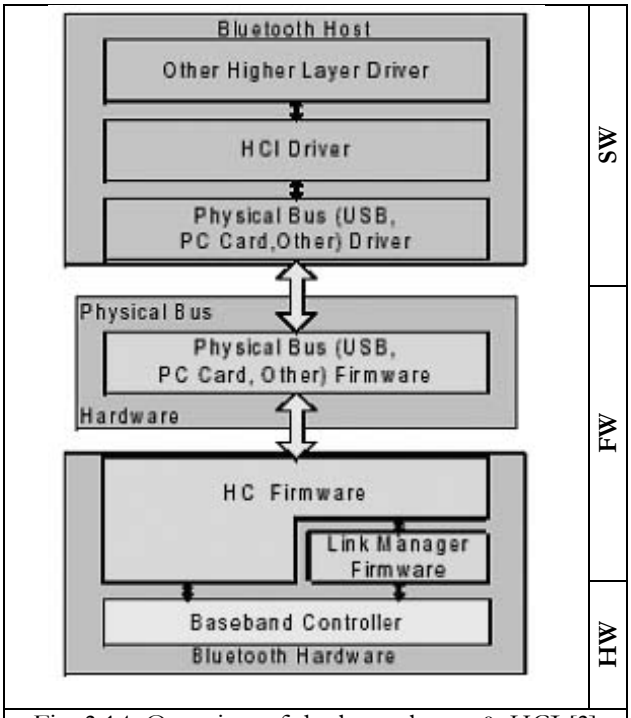


Fig. 3.14: Overview of the lower layers & HCI [3]

[24] I/F: Interface

The traffic crossing the HCI, hereafter referred to as **HCI_PDU**s, comes in three flavors: **command** packets, **event** packets and **data** packets (*L2CAP_PDU fragments of ACL data* or uniform *SCO data*). The *HCI commands* are issued from the host and pertain to all functions of the Bluetooth module such as setting operational parameters, configuring the module's operational status, reading and writing specific low-level registers etc.. The HC notifies the host of the outcome of a command with an *HCI event* either soon after the command is issued from the host or at a later time when a specific action has (or has not) taken place. The reason that HC transmissions to the host are called events instead of responses is that the HC may initiate its own request or send a transmission to the host without the host's prior action. The format of commands and events is presented in figures 3.15 and 3.16, respectively. The payload field of a command or an event consists of a number of parameters that vary by command or event type, respectively. Note that, all HCI packets are Little-Endian encoded, as in all Bluetooth bit-fields.

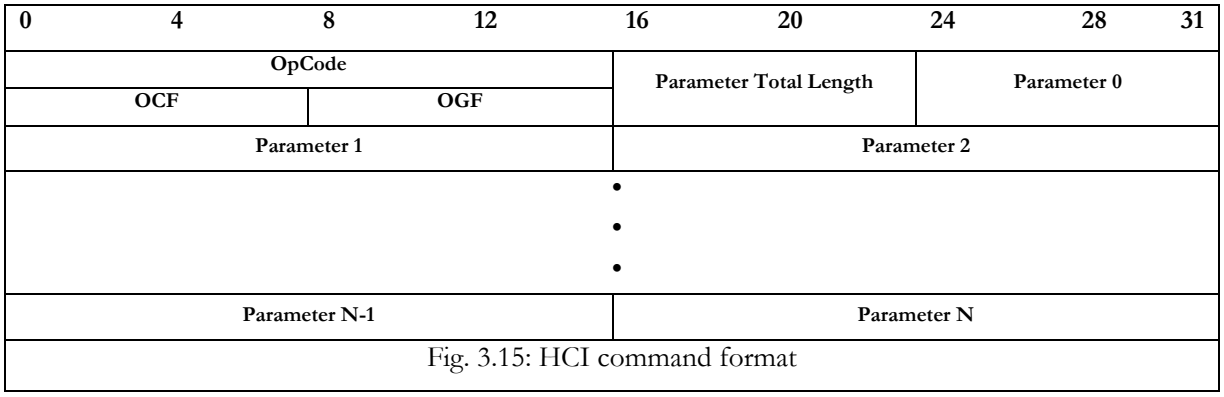


Fig. 3.15: HCI command format

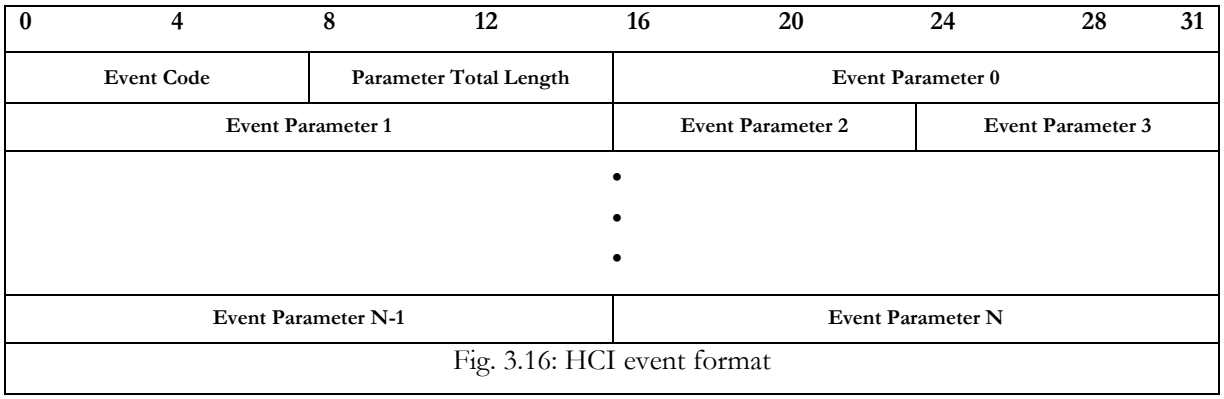


Fig. 3.16: HCI event format

The various fields of a HCI command packet are as follows:

- OpCode: used to uniquely identify different types of commands, with subfields

- OGF[25] (6 bits) identifies the group that the OpCode belongs to:
  - '0b111110' for reserved OGF used for Bluetooth logo testing
  - '0b111111' for a reserved OGF for vendor-specific commands used during module manufacture and testing
  - '0bxxxxxx' for other groups such as link control, link policy, baseband and others (described below)
- OCF[26] (10 bits) identifies a specific HCI command within the particular OGF

- Parameter Total Length: length (in bytes) of the command payload (i.e. length of parameters)
- Payload: the payload of a HCI command is structured as a sequence of variable-size fields for the parameters related to this command

The various fields of a HCI event packet are as follows:

- Event Code: used to uniquely identify different types of events. '0xFE' is reserved for Bluetooth logo-specific events, '0xFF' is reserved for vendor-specific events used during module manufacture and testing
- Parameter Total Length: Length of all of the parameters contained in this packet , measured in bytes
- Payload: the payload of a HCI event is structured as a sequence of variable-size fields for the parameters related to this event

Likewise, the format of ACL and SCO data packets is presented in figures 3.17 and 3.18, respectively.

<table>
<tr><td colspan="8">0 4 8 12 16 20 24 28 31</td></tr>
<tr><td colspan="3">Connection Handle</td><td>PB Flag</td><td>BC Flag</td><td colspan="3">Data Total Length</td></tr>
<tr><td colspan="8">Data</td></tr>
<tr><td colspan="8">Fig. 3.17: HCI ACL data packet format</td></tr>
</table>

[25] OGF: OpCode Group Subfield (identifying a specific group of commands with common characteristics)
[26] OCF: OpCode Command Subfield (identifying a specific command from such a group of commands)

0	4	8	12	16	20	24	28	31
Connection Handle			**Reserved**	**Data Total Length**				
Data								

Fig. 3.18: HCI SCO data packet format

The various fields of a HCI ACL / SCO data packet are as follows:

- Connection Handle: identifies the baseband link (ACL or SCO) over which these data are transmitted or received; connection handles in the range '0xF00' – '0xFFF' are reserved for future use
- Flags:
  - ACL packets:
    - Packet_Boundary_Flag (PB): identifies the beginning ('0b10') or continuation ('0b01') of an upper-layer L2CAP_PDU (as previously seen)
    - Broadcast_Flag (BC): point-to-point ('0b00'), broadcast-to-active-slaves (active broadcast: '0b01'), broadcast-to-all-slaves including any parked ones (piconet broadcast: '0b10'); value '0b11' is reserved for future use
  - SCO packets: reserved for future use
- Payload: data to be carried over the ACL or SCO baseband link, identified by the contents of the Connection Handle field

Transmission of data HCI_PDUs across the physical I/F is regulated by the buffer sizes available on the receiving side of the PDU. Both the host and the HC inquire about the buffer size available for receiving data HCI_PDUs on the opposite side of the I/F and adjust their transmissions accordingly. This implies that a large L2CAP_PDU may need to be fragmented within the HCI layer prior to sending it to the HC. On the receiving side, the HC can reconstruct the L2CAP_PDUs based on the PB Flag information within the received data HCI_PDU. Transmission of HCI_PDUs across the physical I/F follows a FIFO[27] pattern without preemption. Commands and events are processed in their order of arrival but they may complete out of order since each might take a different amount of time to execute. The SAR feature of the L2CAP layer (addressed before, in the "L2CAP" subsection) is the key for fragmenting the L2CAP_PDUs into one or more data HCI_PDUs. Attention must be paid to the

[27] FIFO: First In First Out

previously mentioned fact that SCO (voice) traffic is NOT conveyed through the L2CAP but flows directly through the Baseband layer, instead (fig. 3.19).

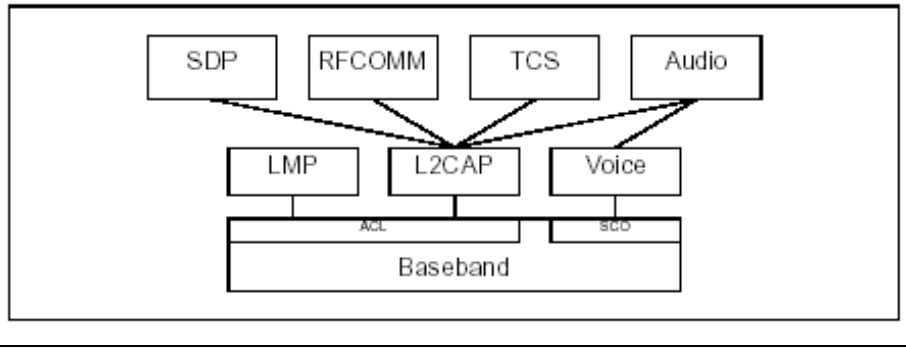


Fig. 3.19: Supported data types of L2CAP layer [3]

The reason is that *direct*, *high-quality* voice with *low delay* and *low overhead* (layer-to-layer propagation delay and additional information shards are avoided) is supported through the Bluetooth stack. So, no SAR functionality is applied to SCO packets and -in this sense- every SCO HCI_PDU is a full, unsegmented SCO packet like the one depicted in figure 3.18. On the contrary, each ACL HCI_PDU is a fragment of an ACL L2CAP_PDU coming through the HCI. This means that in the payload of a typical ACL HCI_PDU a full L2CAP packet structure must exist, like the one presented in figure 3.13. This function is elegantly depicted in figure 3.20 for the HCI running over a USB physical I/F. For simplicity, the stripping of any additional HCI and USB specific information fields prior to the creation of the baseband packets (Air_1, Air_2, etc.) is not shown in the figure. Of course, if the superjacent L2CAP_PDU is small enough to fit inside an ACL HCI_PDU, no SAR occurs. For this to happen, the following comparison must stand[28]:

$$\text{ACL\_HCI\_PDU\_Payload} \geq \text{ACL\_L2CAP\_PDU (= Length + CID + Payload)}$$

Returning to the various types of HCI commands, it should be said that various groups are defined with respect to the specific set of operations they encompass. These groups are distinguished by the OGF subfield of the OpCode and for each HCI command there is a corresponding HCI event bearing the

[28] This detail in the implementation of the Bluetooth stack is somewhat hard to discover in the BT spec. but is a key element to the deployment of this thesis and will be met again in chapter 6.

outcome and the return parameters of the operation triggered. The groups called upon in the HCI specification are the following:

- *Link Control commands*: contains commands for inquiry initiation, ACL/SCO link setup & termination, authentication/encryption initiation & configuration, clock offset & user-friendly name of remote device querying etc. (OGF: '0b000001')
- *Link Policy commands*: contains commands for power-management policy, QoS parameters passing from L2CAP to LMP layer, role switch etc. (OGF: '0b000010')
- *HC & BB commands*: contains commands for accessing HW registers of the module, inquiry/page scan (de)activation, parameter configuration for authentication/encryption, flushing of ACL/SCO data packets pending transmission etc. (OGF: '0b000011')
- *Informational Parameters*: contains commands that request static information about the HW and FW that is hardwired to a Bluetooth module, e.g. request for current version of various protocols like HCI, LMP etc. (OGF: '0b000100')
- *Status Parameters*: contains commands that request static information that is dynamically updated like the value of a contact counter holding the time between the response of the remote device to a flush of transmitted data or the quality of a specific link by measuring its signal strength etc. (OGF: '0b000101')
- *Testing commands*: contain commands that provide the ability to test various functionalities of the Bluetooth HW. They also provide the ability to arrange various conditions for testing (OGF: '0b000110').

It must be noted that none of the fields in any of the HCI_PDUs identifies the HCI_PDU type: command, event, ACL data or SCO data. The identification is left to the HCI transport protocol that actually carries the PDUs between the host and the HC. Strictly speaking this is a violation of protocol layering; however, it allows the HCI to take advantage of the capabilities of the underlying transport protocol (e.g. RS-232, USB) which may provide its own means for distinguishing the four HCI_PDU types with minimal overhead. For more technical information on the subject, please see chapter 4. Additional insight and more details regarding the HCI layer will be presented -if needed- in chapters 5 and 6 should they prove fruitful in explaining some implementation choice or technique used. Hereon, a brief overview of the remaining protocol layers of the Bluetooth stack will follow.

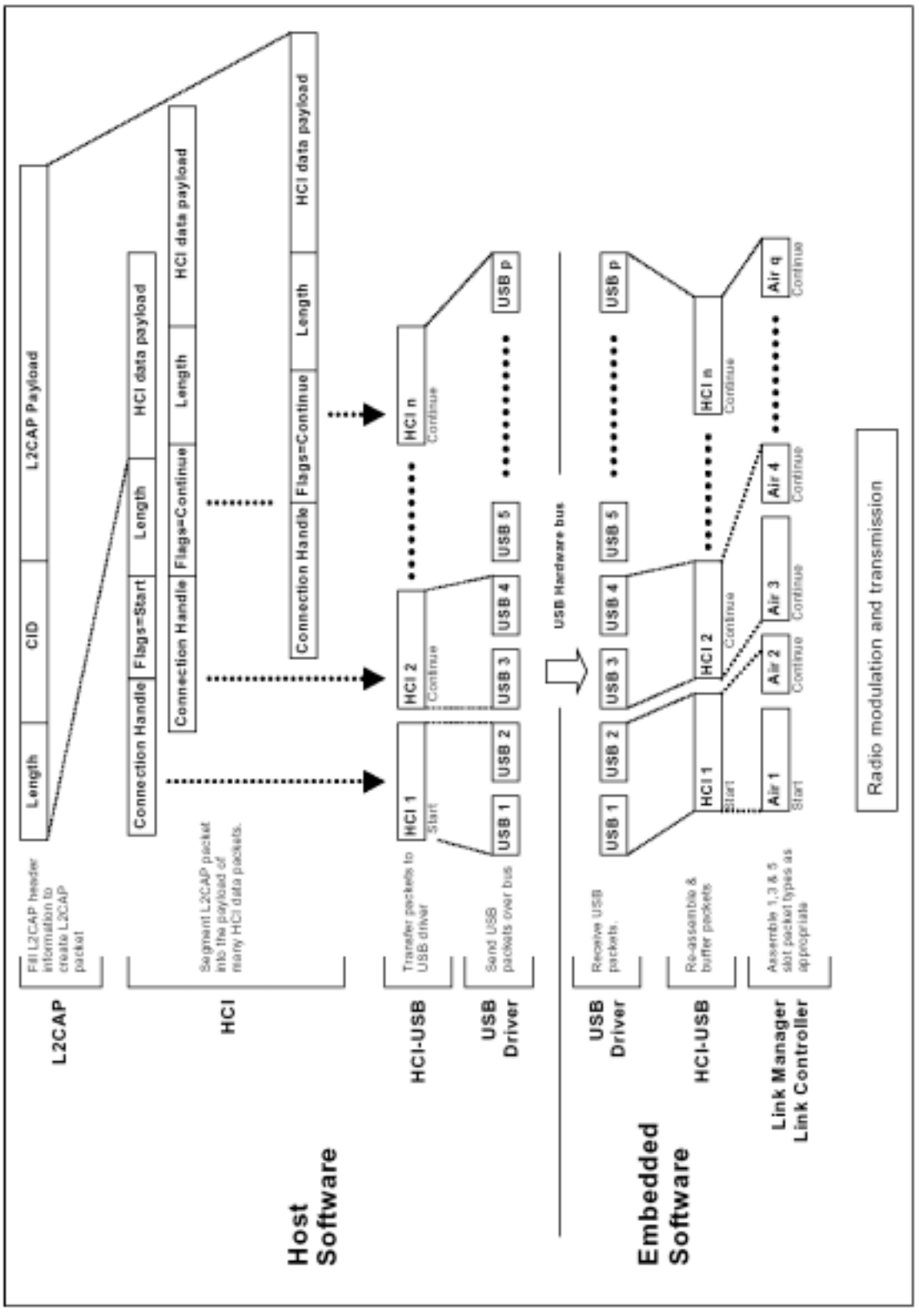


Fig. 3.20: SAR services in a system with a HCI layer [3]

3.2.6 RFCOMM Protocol

Serial I/Fs are ubiquitous in computing and telecommunications devices. Because Bluetooth technology aims at replacing cables, it seems clear that there is a large opportunity to replace serial cables. Yet, the transport group protocols are not modeled after a serial port. Thus, the SIG has chosen to define a layer in the protocol stack that emulates the typical serial I/F: the RFCOMM layer which is currently the basis for most of the BT profiles. The name "RFCOMM" connotes a wireless (RF) instance of a virtual COM port. This protocol layer has been gravely based upon the ETSI TS 07.10 [ETSI99] standard which is a good match for the needs of the Bluetooth technology. By modifying the standard –by throwing away the abundant overhead-, specific features were defined to match the Bluetooth functionality. The RFCOMM supports **multiplexing** of multiple simultaneous clients-applications (maximum number of 60 per RFCOMM connection) running on top of it (applications group) and is **RS-232 compatible** supporting all 9 signals of the RS-232 I/F: Signal Common, Transmit Data (TD), Received Data (RD), Request to Send (RTS), Clear to Send (CTS), Data Set Ready (DSR), Data Terminal Ready (DTR), Data Carrier Detect (CD), Ring Indicator (RI).

RFCOMM uses an L2CAP connection (CO channel) to instantiate a logical serial link between two devices. Only a single RFCOMM connection is permitted between two devices at a given time. The first RFCOMM client establishes the RFCOMM connection over the L2CAP; additional users of the existing connection can use the multiplexing capabilities to establish new channels over the existing link. Each multiplexed link is identified by a unique 6-bit number called **Data Link Connection Identifier** (**DLCI**).

3.2.7 Service Discovery Protocol (SDP)

The SDP defines how a Bluetooth client's application shall act to discover available Bluetooth servers' services. It defines how a client can search for a service without knowing anything of the available services. The SDP provides means for the discovery of new services becoming available when the client enters an area where a Bluetooth server is operating. It also provides the means to detect when a service is no longer available. This protocol layer is of utmost importance for the Bluetooth technology because of its dynamic, ad hoc nature. Even thought the service discovery concept is not new and some standard might be adopted or modified like in the case of RFCOMM, the SIG has developed its own unique SDP which is optimized for Bluetooth wireless communication. Key features sought in the SDP are:

- Simplicity: SDP is part of nearly every usage scenario so it should be simple ("make the common function fast")
- Compactness: SDP is a typical operation in most cases of link establishment, so it should not be very time-consuming
- Versatility: SDP has been defined broadly and somewhat "loosely" in order to be easily extended to include all future usage scenarios and profiles
- Service location by class & by attributes: Client devices must easily find a requested service in an ad hoc network searching for a specific class of service (e.g. printer), a specific attribute of the service (e.g. color printer) and a specific instance of a service (e.g. a specific physical printer)
- Service browsing: In addition to searching for services by class or attribute, it is often most useful to browse the available services for the ones that look interesting.

3.2.8 Telephony Control Protocol (TCS-BIN)

The telephony control protocol is embodied by the TCS-BIN layer; it is based upon the existing ITU-T Q.931 protocol [ITU98] and constitutes a binary encoding for packet-based telephony control residing over the L2CAP layer. Particularly, TCS-BIN is used for the **call control** aspects of telephony including establishing and terminating calls along with more control functions. It can be used to control both voice (via SCO packets) and data (via ACL packets) calls. TCS-BIN also defines a method for devices to exchange call signaling information without having a connection established between them (also known as **connectionless TCS**).

3.2.9 Audio

Bluetooth audio has been extensively covered, especially in the "Baseband" and "HCI" subsections. However, some basic traits are given to make the picture fuller. The transmission rate for Bluetooth audio traffic is set at 64 Kbps, chosen to be sufficient for normal voice conversations. While the communication of other audio media (e.g. music) over Bluetooth audio links is not precluded, the design is not based upon such audio traffic; it clearly is centered on voice traffic. Two types of encoding schemes are specified for Bluetooth audio: **Pulse-Code Modulation** (**PCM**) with either of two types of logarithmic compression (**A-law** or **μ-law**) applied; also, **Continuous Variable Slope Delta** (**CVSD**) modulation. Probably, the modulation presenting best results for voice traffic is CVSD.

3.2.10 Non-Bluetooth-specific protocols

Hereon, a brief presentation of the remaining protocols of the Bluetooth stack is made. Presenting them in full extent would miss the goal of this thesis; however they are key protocols to supporting and boosting the device interoperability attempted through Bluetooth technology.

3.2.10.1 Telephony Control – AT Commands

Bluetooth supports a number of AT commands for transmitting control signals for telephony control through the serial port emulation (RFCOMM).

3.2.10.2 Point-to-Point Protocol (PPP)

The PPP is a packet-oriented protocol and must therefore use its serial mechanisms to convert the packet data stream into a serial data stream. It runs over RFCOMM to accomplish point-to-point connections. In the BT spec. the subject of WAP client-server communication over PPP is deployed (see below).

3.2.10.3 UDP – TCP/IP Protocols

The UDP/TCP and IP standards allow Bluetooth units to communicate with other units connected, for instance, to the Internet. Therefore, the Bluetooth unit can act as a bridge to the Internet. The TCP/IP/PPP protocol configuration is used for all Internet Bridge usage scenarios and OBEX (see below) in future versions. The UDP/IP and PPP configuration is available as a transport to WAP.

3.2.10.4 Wireless Application Protocol (WAP)

The WAP is a wireless protocol specification that works across a variety of wide-area wireless network technologies bringing the Internet to mobile devices. Bluetooth can be used like other wireless networks with regard to WAP to provide a bearer for transporting data between the WAP client and its adjacent WAP server. Furthermore, Bluetooth's ad hoc networking capability gives a WAP client unique possibilities regarding mobility compared with other WAP bearers. Also, the server push capability of the WAP technology opens new possibilities for distributing information to handheld devices on location basis, if used over Bluetooth.

3.2.10.5 Object Exchange (OBEX) Protocol

IrOBEX (OBEX, for short) is a session protocol that is adopted by the **Infrared Data Association** (**IrDA**) in an attempt to achieve IrDA-stack interoperability given that infrared-enabled devices are currently very wide-spread[29]. OBEX is an optional application layer protocol operating over the RFCOMM and designed to enable units supporting infrared communication to exchange a wide variety of data and commands. It uses a client/server request-response model and is independent of the transport mechanism and transport API. Note that, the provided interoperability resides ONLY at the application layer and not at the physical layer (i.e. a Bluetooth module cannot communicate directly with an infrared transceiver over the air).

Having traversed the whole Bluetooth stack, the Bluetooth profiles created for this wireless technology (and some major usage scenarios stemming from them) arise. A brief description of those profiles and usage cases is included in Appendix B, since they are not part of the thesis scope, strictly speaking. For more details on the Profiles, refer to [4].

3.3 Bluetooth in a wireless-crowded world

Before ending this chapter and having taken a sufficiently thorough look at the Bluetooth architecture, its potential and its usage scenarios, a brief comparison with other dominant -at the time- wireless protocols can take place. Along with the development of the Bluetooth protocol, many other wireless solutions have emerged or already constitute mature technologies. One such technology is IEEE 802.11 protocol and an alphabet soup of versions: 802.11b (also known as Wi-Fi[30]), 802.11a (Wi-Fi5) and 802.11g; also the two 802.11a counterparts HyperLAN and HyperLAN2 and another protocol of proportional features to 802.11, HomeRF. Also, belonging to the small range wireless systems, a most popular member of the wireless family and widely used these days is the IR protocol. A brief comparison of Bluetooth against 802.11 and IR technical characteristics follows.

[29] Actually, apart from IrOBEX, the IrOBEX associated data object formats are also adopted, as well as the Infrared Mobile Communications (IrMC) method of synchronization.
[30] Wi-Fi: Wireless Fidelity

3.3.1 Bluetooth vs 802.11

Bluetooth and 802.11 are both wireless 'network' technologies. Yet, from the early beginning the major field of application for IEEE 802.11 has been the creation of wireless access points and -in so doing- the realization of WLANs. This should be no hard task for the protocol since it uses the same upper OSI layers as the known wired protocols (TCP/IP, Ethernet etc.). Both technologies operate in the same 2.4 GHz ISM band and are capable of creating ad-hoc networks. The major difference is the one in the data rate of the two technologies. Bluetooth runs at a much slower data rate than 802.11. Bluetooth has a maximum capacity of 1 Mbps whereas the, now standard, 802.11b runs up to 11 Mbps (The IEEE 802.11 standard also only ran at 1 Mbps). The reason for the different data rates between the two technologies lies in the Physical and Data layers. The physical layer ("Radio") of Bluetooth has very little transmitter power at the antenna, as opposed to the high output power of an 802.11 transceiver. The output power of a typical Bluetooth transmitter is 1 mW whereas the output power of 802.11 is 1 W (i.e. a factor of 1000 in the output power of the transmitters). From this difference stems the operating range of the two different communication systems: 10 m for Bluetooth and 100 m for 802.11b. This gives 802.11 users much more flexibility in using their devices. Also, the modulation technique has something to do with the data rate also. Bluetooth uses GFSK (Gaussian Frequency Shift Keying) as opposed to CCK (Complementary Code Keying) in 802.11. In terms of transmission pattern, Bluetooth uses FHSS while 802.11b uses DSSS. Frequency hopping could cause delays in the transmission and the only way to prevent this is to slow down the information exchange. In the DSSS case, the CDMA allows the signal to be spread out over a large frequency range and make all other users look like noise to the destination. This allows for higher data rates and more users. Another difference is the usage. As stated before, Bluetooth is being used for device to device data transfer (and voice). This allows devices such as PDA's, notebooks, cell phones etc. to talk to each other, on the fly. The use of 802.11 has become more of that of a wireless access point for a computer to get on a wired backbone. This is the direction that the developers have been taking in the past few years. 802.11 does have the ability to create ad-hoc networks, but that isn't the way many people have decided to use it. As far as security is concerned, Bluetooth seems -at least for now- a tougher player in the field with its 128-bit encryption patterns and the FHSS scheme to make packet "sniffing" a difficult task due to the persistent frequency change. On the other hand, 802.11 uses a security protocol called Wired Equivalent Privacy (WEP) which uses 64- or 128-bit encryption and -by many accounts- is not as

robust or effective a LAN security protocol as it was initially presented. Additionally, since it is concerned with WLANs, packets are distributed to anyone in range and the task of security is more difficult. Also, due to its frequent use in wireless access points which act like hubs connecting to a wired network, IEEE 802.11 cannot have much security at the physical layer, one way or the other.

3.3.2 Bluetooth vs IR

IrDA is used for high-speed, short-range, line-of-sight (LOS) and point-to-point data transfer. The range of IrDA is larger than 1 m. It requires a narrow angle (30^0) point-and-shoot operation. The maximum data transfer rate is 4 Mbps and 16 Mbps rate is under development. It doesn't interfere with other wireless communications and is also immune to interference from others. IrDA gained great acceptance worldwide. Currently over 150 million units are installed worldwide and this number is growing 40% annually. Its major applications are laptop computers, printers and LAN access among others. The biggest advantage of IrDA over Bluetooth is its high throughput, which makes it suitable for high-speed applications. The $5-solution attempted by the companies in the Bluetooth market is remarkable; yet, the IrDA is even cheaper. One manufacturer can get a whole solution with cost of about $1. This low-cost attribute is gravely boosted by the mass production that IR-enabled devices undergo. Bluetooth, however, allows greater mobility; as previously discussed, for class 2 Bluetooth devices, transmission range can reach 10 m, it is omni-directional (as opposed to the LOS attribute of IR) and can effectively penetrate clothes and soft partitions which the IR beams cannot; this is due to the difference in the wavelength and -ergo- in the behavior between RF signals and IR signals.

3.3.3 Asking The Right Question

Since the appearance of all the above wireless technologies, the question whether Bluetooth is to be shadowed by them is a common one. Many articles have been posted and long talks in various internet message boards, newsgroups and chat rooms have taken place on this subject. However, the above question appears to be a totally misplaced one. The reason is that Bluetooth is not a rival of such technologies but rather a *complementary* technology to them. The difference in the operations sought and the implementation technology separates the fields of application to a great extent creating no ill rivalry among the various technologies. In Appendix A, figure A.5 proves the truth of this statement by presenting a cumulative list of those technologies.

The major issue here is not to get blindly mingled in some speed race about wireless technologies but -in an extremely wireless-crowded world as this- to work towards enabling the coexistence of such diverse wireless technologies. Wi-Fi and Bluetooth operate in the unlicensed 2.4 GHz unlicensed spectrum and as the activity of both technologies continues to grow, Wi-Fi networks will operate in the presence of increasing amounts of Bluetooth signals in enterprise, home, and public area environments. Users will want to use whatever wireless devices surround them, when they want to, and how they want to. Because the two technologies share the same 2.4 GHz band, there is the potential for interference. When devices are beyond one meter in range, there is typically "graceful degradation". However, when the devices are within a meter, and particularly when the technologies are co-located, as in the case of a notebook computer, the interference can be severe. In these cases, interference can result in noticeable performance degradation or even total lack of applications support. For this reason the need for multi-standard radio technology is becoming more imminent; furthermore, wireless infrastructure in the vastly-crowded unlicensed bands (like the 2.4 GHz ISM band) will require robust, interference tolerant system designs. Currently measures are taken and industry standard bodies such as IEEE 802.15.2[31], the Bluetooth SIG, and the IEEE Coexistence TAG[32] have recognized the above needs working diligently to deliver the seamless connectivity that will empower the wireless generation [10], [11].

31 In late 1999, IEEE's 802.15 Working Group created Task Group 2 (TG2, also called 802.15.2), whose mission is to develop recommended practices for coexistence between WPAN (Bluetooth) and WLAN (802.11b) technologies.

32 Coexistence Technical Advisory Group (Coex TAG): its role is to recommend how best to create a formal review process for coexistence of wireless standards within IEEE; the goal is to insure that sensible coexistence policies and techniques are brought to the market along with the technologies.

4

Application & Training Tool Kit

4.1 Getting Started

The "Applications And Training Tool Kit" is produced by Teleca Comtec [32] under Ericsson Licensing. It is a SW/HW suite aimed at developing various Bluetooth-centered or Bluetooth-enabled applications. It consists of a circuit board driving a Bluetooth module which implements (in HW) the lower transport protocols up to the HCI layer and of an accompanying SW packet (**Bluetooth PC Reference Stack**) implementing part of the MW protocols. The disclaimer accompanying the Tool Kit states that it is a product for development and/or demonstration purpose only and it has NOT been formally tested for compliance with the Bluetooth specifications. Qualification for the Tool Kit is based upon a declaration of compliance with the BT spec. v1.0b (plus critical errata) and is listed as a qualified product on the Official Bluetooth Website. For technical reasons it must be noted that this module is NOT supported by Ericsson but by Teleca Comtec alone.

4.2 Hardware Components

The HW consists of a two-layer PCB[33] (fig. 4.1) equipped with a UART buffer, a voltage regulator, a few passive components and the Bluetooth Module of Ericsson communicating with the outside world through a proper HCI over one of two supported physical I/Fs: UART and USB. It also bears a PCM I/F for voice data but no codec circuit is included. The USB I/F is high-speed (12 Mbps), compliant with USB Specifications 1.1. When using the USB interface, the module appears as a USB slave device and therefore requires no PC resources. The Bluetooth Module (ROK101 008) includes the Ericsson Baseband device, a Flash Memory and the Ericsson Radio Module device and has been included in various designs like the EBDK presented in chapter 2. If UART is selected, the communication is assumed to be free from line errors. Four signals will be provided on the UART. TxD and RxD are used

[33] PCB: Printed Circuit Board

for data and RTS and CTS are used for flow control. The module is DCE (Distributed Computing Environment). The PCM data (externally encoded) can be:

- PCM, 13-16 bit
- μ-Law 8 bit
- A-Law 8 bit

The PCM sync is 8 kHz and the PCM clock 200 kHz - 2 MHz. (These features are inherently supported by the Bluetooth ROK101 008 chip, as seen below).

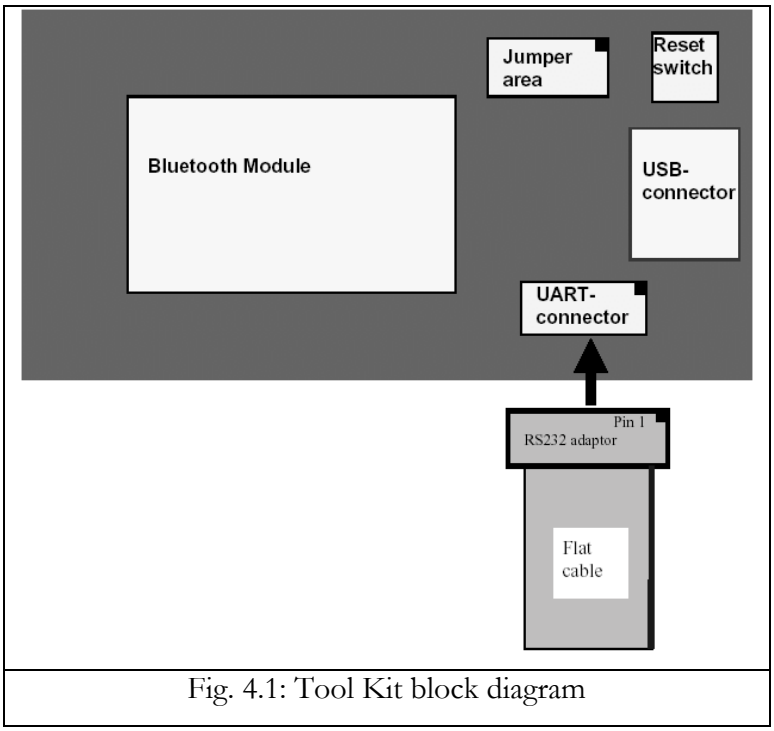


Fig. 4.1: Tool Kit block diagram

The first decoded HCI command received by the Bluetooth module through either of the USB or UART ports determines which port to be selected. The other one is switched off, until next HW reset. The kit supports PC power management as outlined by the ACPI specification and is compatible with all ACPI-compliant operating systems. This includes support for notebook system wake-up and ACPI PMI event generation. Two side-band signals Wake-up and Detach are used to augment control of the state from which the notebook resumes. When the host is in a power down mode, Wake-up wakes the host up when the Bluetooth system receives an incoming connection. The host indicates that it is in Suspend mode by using the Detach signal. To make easy access possible to certain signals, a jumper area is included on the Tool Kit. The signals in this jumper area and in the other two board connectors

(UART, USB connectors) are listed in figures 4.2 and 4.3, respectively. Designators within brackets refer to the pin description for the Bluetooth Module in its data sheet (examined below). PCM enables voice use together with an external codec system, but there is no support for this implementation in the Tool Kit. Note that pin 1 of the connectors is marked with a square dot in the previous layout picture [13]. Also, the Tool Kit includes a 5 ohm inverted-F antenna on-board, connected to the Bluetooth Module which -thus- a Class-2 device.

9 RESET (R3)	**7** PCM_IN (A1)	**5** PCM_OUT (A2)	**3** WAKE_UP (B4)	**1** Vin 5V nom
10 GND	**8** DETACH (C1)	**6** VCC 3.3V * (C2, C4 and C6)	**4** PCM_SYNK (A3)	**2** PCM_CLK (A4)

Fig. 4.2: Tool Kit jumper area

9 GND	**7** not used	**5** TXD (B5)	**3** RXD (A5)	**1** not used
10 not used	**8** not used	**6** RTS (A6)	**4** CTS (B6)	**2** not used

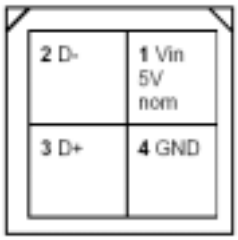


Fig. 4.3: Tool Kit UART (left) & USB (right) connector

If the USB I/F is used, then no external power supply is required (through the USB cable). If an alternative to the power supply through USB is preferred, the requirements are listed below:

- Supply voltage: min +4.4 V, max +5.25 V connected to Jumper area pin 1 (relative GND pin 10)
- Minimum supply current: 100 mA

The heart of the above seen circuit board definitely is the Ericsson Bluetooth Module [13], modeled ROK101 008 (ROK chip, for short). In fact the circuit board is there to drive and provide the connectors to the I/Fs and various functionalities of the ROK chip. The module consists of three major parts; a baseband controller, a flash memory, and a radio that operates in the globally available 2.4–2.5 GHz free ISM band. The functionality of the ROK chip extends beyond the handles given by the Kit for it, meaning that not all chip features are accessible externally. In this sense, both data and voice transmission is supported by the module.

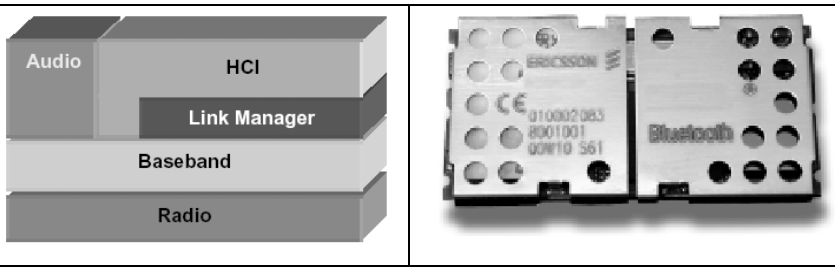


Fig. 4.4: ROK101 008 HW & FW stack (left), physical view (right)

Communication between the module and the host controller is carried out via UART and PCM interface. It is a very powerful and highly integrated circuit with the following key features:

- RF output power class 2
- FCC and ETSI approved
- 460 kb/s max data rate over UART
- UART and PCM interface
- I2C interface
- Internal crystal oscillator
- HCI firmware included
- Point to Point (PtP) connection
- Built-in shielding

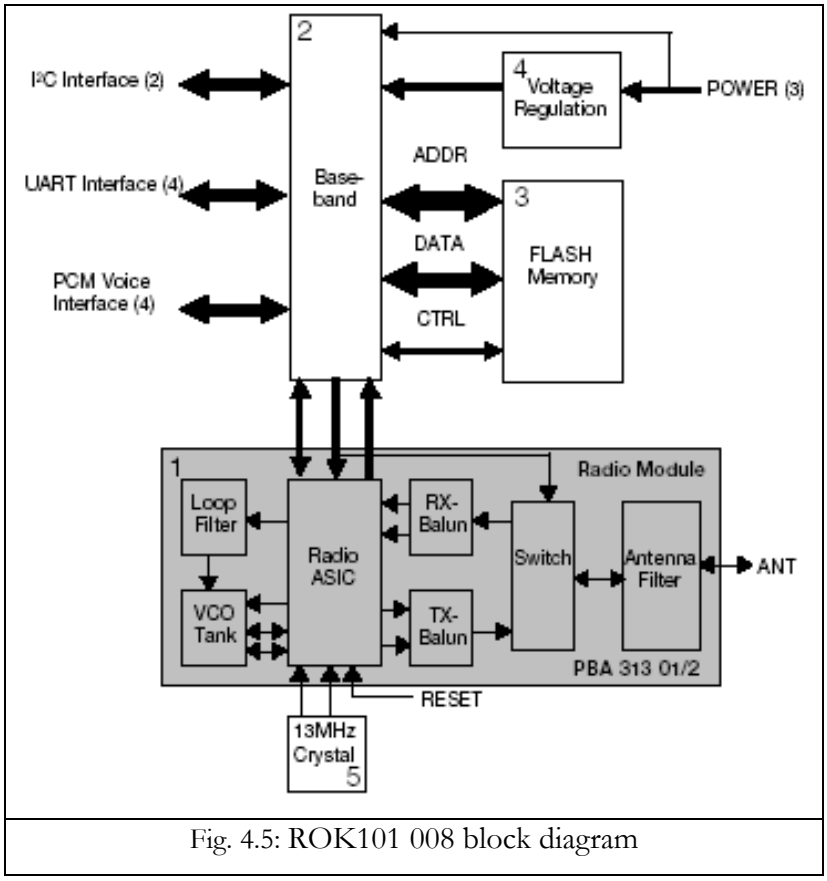


Fig. 4.5: ROK101 008 block diagram

The Bluetooth protocol layers implemented (HW & FW) are depicted in figure 4.4 while the block diagram[34] of the chip is depicted in figure 4.5. The module includes FW for the HC interface, HCI, and the LM. This FW resides in the Flash. The UART implemented on the module is an industry standard 16C450 and supports the following baud rates: 300, 600, 900, 1200, 1800, 2400, 4800, 9600, 19200,

[34] For more details on the operational blocks and specific functionality of the ROK chip, see [13].

38400, 57600, 115200, 230400 and 460800 bps. 128-byte FIFOs are associated with the UART. Four signals will be provided for the UART interface. TxD & RxD are used for data flow, and RTS & CTS are used for flow control. The standard PCM voice I/F has a sample rate of 8 kHz. The PCM clock is variable between 128 kHz and 2.0 MHz in the PCM slave mode. The PCM data can be linear PCM (13-16bit), μ-Law (8bit) or A-Law (8bit). The PCM I/F can be either master or slave. Over the air, the encoding is programmable to be CVSD and A-Law or μ-Law. A master I2C I/F is available on the module. The control of the I2C pins is performed by Ericsson specific HCI commands available in the FW implementation.

Apart from those, there are some added Ericsson-specific HCI commands for other functions e.g. directly writing the HW registers of the module, changing the UART speed etc.. More details will be presented in chapters 5 and 6, if required. As stressed out in the previous chapter, HCI does not provide the ability to differentiate the four HCI packet types. Therefore, if the HCI packets are sent via a common physical I/F, a HCI **packet indicator** has to be added. Since the I/F used in this thesis is the UART I/F, the indicators provided by the BT spec. and -of course- supported by the ROK chip are presented in figure 4.6. One such indicator precedes *every* packet transmission to and from the HCI layer, depending on the HCI packet type. More details on the usage of those indicators and HCI packet synthesis are presented extensively in chapter 5.

Before concluding with the HCI it is imperative to list those features NOT supported by the version of the ROK chip at hand; this is merely a matter of the included FW version and is subject to change soon. Concisely, there is no support for:

- all connection power management modes (hold, sniff, park)
- master-slave role switch
- timing accuracy
- slot offset
- 5-slot packets
- 3-slot packets
- HV2 packets
- channel quality-driven data rate
- transparent SCO data

HCI packet type	HCI packet indicator
HCI Command Packet	0x01
HCI ACL Data Packet	0x02
HCI SCO Data Packet	0x03
HCI Event Packet	0x04

Fig. 4.6: HCI packet indicators

- power control
- paging scheme

Most of the above list has been provided by issuing the Read_Local_Supported_Features HCI command to the LM and should be considered in technical matters. Attention must mainly be paid to the fact that the chip does not support multipoint connections (i.e. only one connection can be up and running at any time).

4.3 Included Software

The *Bluetooth PC Reference Stack* is a PC-ported version of the Bluetooth Host Stack (in C++), which is a SW component developed by Ericsson to enable local wireless connections between devices such as mobile computers, handheld units, mobile phones, LAN access points, digital cameras headsets etc. This Bluetooth Host Stack is independent of operating system (OS) and HW. The Bluetooth Host Stack complies with the Bluetooth Specification and is used in many Ericsson products. It is a Qualified pre-tested Component with the Covered Functionality of: *HCI Driver*, *L2CAP*, *RFCOMM*, *SDP* and *OBEX*. The Bluetooth PC Reference Stack includes the executable of a COM-server, containing the Bluetooth Host Stack (along with the OBEX component) and communicates with the Ericsson Bluetooth Module (ROK chip) on the Tool Kit via the HCI I/F.

There are two sample applications with source code on the accompanying Tool Kit CD:

1. A sample application (TestSample) suite that contains 3 separate applications that work together as an application over RFCOMM, showing the basic use of the Stack. This application can be used as an example when writing non-windows based programs on top of the Stack. This PC-reference Stack delivery is then used only as a development environment for other environments. It demonstrates connect/disconnect of a headset profile (with source code) and presents the message flow towards the stack API, and it does not take care of the actual voice transfer.
2. A sample application (Chat) suite that also contains 3 separate applications that work together as an application over RFCOMM, showing the basic use of the Stack. This application can be used as an example when writing windows-specific programs on top of the Stack. It demonstrates a chat application (with source code).

Those sample applications utilize the COM server, which includes the Bluetooth PC Reference Stack (source code NOT included).

More data could be provided for the PC Reference Stack but this SW suite -holding source code and executables- is NOT used in this thesis. The obvious reason is that -as stated at the very beginning, in chapter 1- the goal sought in this thesis is to build an *embedded applications environment* for handling the Bluetooth module and to garnish it with a specific application pointing out its functionality. The "embedded" element makes the utilization of all or any part of the C++ source code provided with the Tool Kit impossible. The SW axis of movement is rather (as seen in chapter 5) AVR *assembly* language, well-suited for building such an *embedded* system. Yet, some insight on the rationale of implementing (in SW) part of the host for this embedded design was gained by studying those source codes. Also, the possibility of using an appropriate C++ compiler for converting the whole or part of the PC Reference Stack (structured in C++) into AVR assembly should not be easily overlooked, yet, has not been exploited in this thesis. For more details on the structure, APIs etc. of the PC Reference Stack, see [15].

4.4 Package remarks

Before concluding this chapter, it should be stressed out that all the above mentioned characteristics are very rapidly changing, creating a somewhat blurry picture. The Bluetooth Training & Applications Tool Kit at hand is a package including the **toolkit CD** (ver. R2A), the **PC Reference Stack** (ver. R1C) and the **ROK 101 008 chip** (FW ver. R1A). All these versions are very confusing and are persistently replaced by newer ones; this makes it hard to keep track of all the new features. Should a new version of the Kit be used, the version number of every component must be thoroughly checked in order to discover potential design changes, compatibility problems etc..

5

Bluetooth Applications Environment

5.1 System Overview

The Bluetooth Applications Environment (**BlueApplE**) is a host platform built in this thesis with the purpose of fully controlling a Bluetooth Module. The heart of the whole design is a microcontroller which we have seamlessly incorporated in the design and which constitutes the "mind" behind all aspects of system functionality. In this chapter, a complete description of the structure and capabilities of the BlueApplE is attempted. A clear-cut framework of the design will be given, emphasizing on both the HW and SW aspects of the system. Note, however, that details of the application deployed on the BlueApplE system, the so-called BlueBridge, will be kept to a minimum for the time being, so as to make things less perplexing; the application will be thoroughly examined in chapter 6.

5.2 Hardware Specifics

A functional block diagram of the BlueApplE is depicted in figure 5.1. Four basic blocks can be distinguished. The system depicted below is fully symmetrical in the sense that it is built in two copies, one copy per available Bluetooth Module. The **Bluetooth Module** is -by far- the most essential part of the design since it is the one that enables Bluetooth technology. This module is the one described in the previous chapter, the "Applications & Training Tool Kit", incorporating the Radio, Baseband, Link Manager and HCI layers in a single chip along with a low-power, short-range on-board antenna. Yet, this Ericsson-distributed board and chip include no CPU for controlling the fragment of the Bluetooth stack. This must be done by a separate host residing on the other side of the HCI. This brings discussion to the next most important part of the system, the host.

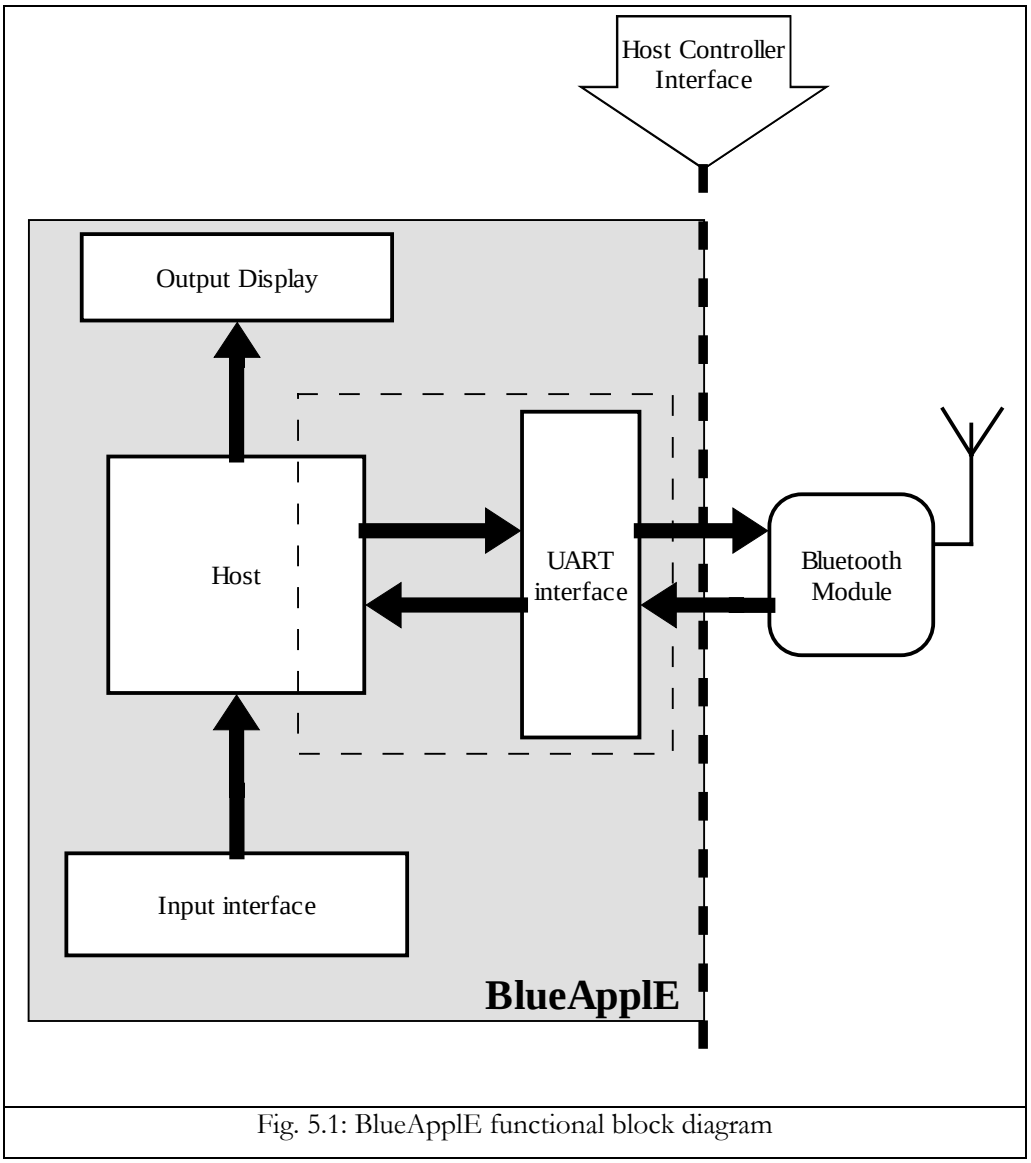


Fig. 5.1: BlueApplE functional block diagram

The **Host** is the "mind" of the design; a CPU that regulates all control and data flow to and from the Bluetooth module. Obviously, the host is a sine-qua-non element for making the BlueApplE system an embedded one. As seen in the previous chapter, the HCI layer is substantiated upon two available physical transports, a USB and an RS-232 I/F. The one used in the BlueApplE is RS-232. The choice was based on the following fact: USB is a technology far more fresh and advanced than serial communication but it is also far more complicated to implement, especially in embedded systems such as the one at hand. Suffice to say that, even though in terms of physical connections it is rather simple (only four signals are utilized, see fig.4.3), its operation is based on a huge SW stack which is difficult to implement, let alone squeezing it into an embedded design as this. Developing an embedded USB stack

for the system would be a thesis of its own. A token of the task's difficulty is that -only lately- great companies like Atmel have managed to fit a meaningful part of the USB stack inside a microcontroller [16]. Should the host reside in a PC, the above task would be far easier to implement; what is more, many OS drivers exist now for transparently driving a USB device (providing user-friendly I/Fs and handles) or for developing custom applications. Even though USB is multi-featured (stream multiplexing, power management etc.) and faster than RS-232 (11 Mbps vs. 460.8 kbps), the latter provides an easy, direct and less time-consuming means of conveying data between the host and the Bluetooth module. Keeping transfer lines (RX, TX) as short as possible and making careful connections has enabled an error-free transmission; in fact, the physical transport used is the UART subset of RS-232 in order to avoid additional cost in time to implement flow control inside the host (given the Bluetooth module inherently supports flow control over UART). In the absence of control signals (RTS, CTS) the design has indeed worked uninterruptedly during all development time.

The last two major functional blocks of BlueApplE are the **Input Interface** and the **Output Display**. BlueApplE has been designed in such a manner that the potential user can operate it with little or more knowledge of the underlying architecture and of the Bluetooth functionality. The Input Interface is a subsystem designed to give this kind of functionality to the external user. It provides basic functions that hide all low-level details making communicating to the Bluetooth module an easy task while -at the same time- it provides a convenient and direct access to internal values for micro-trimming the system resources and operations. As seen later on, this subsystem is easily reconfigurable to support more or different operations with minimal effort. The Output Display, as its name proclaims, is the subsystem displaying the state and active operations of BlueApplE at any time. It is responsible for all error and state messages; also for concurrently displaying selections made by the Input Interface. All of the above blocks (along with the "UART interface" block) will be revisited in details, in the following section.

5.2.1 The Host

It is now time to shed some light at the HW details of the design, starting with the most important component, the host. Trying to combine the features of i) component availability, ii) maximum system performance and iii) low cost, the module selected to play the part of the host CPU has been AT90S8515, the 8-bit AVR RISC microcontroller (hereon, mC) by Atmel [17]. Although this is a well-known mC, its basic features will be briefly cited; this will help later on, during the tying of the mC

various functions to specific tasks of the embedded system. In short, AT90S8515 (fig. 5.2) provides the following features:

- 8 Kbytes of In-System programmable Flash
- 512 bytes EEPROM
- 512 bytes SRAM
- 4 x 8 general-purpose I/O lines, - *Port A, Port B, Port C, Port D*
- 32 general-purpose working registers (high 16 registers are capable of immediate operation), - *r0 to r31*
- 2 timer/counters (8-bit & 16-bit) with compare modes, - *T/C0, T/C1*
- 2 external (- *INT0, INT1*) and 10 more internal interrupts
- programmable serial UART
- programmable Watchdog Timer with internal oscillator
- SPI serial port
- 2 software-selectable power-saving (sleep) modes:
    - Idle mode: stops the CPU while allowing the SRAM, timer/counters, SPI port and interrupt system to continue functioning
    - Power-down mode: saves the register contents but freezes the oscillator, disabling all other chip functions until the next external interrupt or HW reset

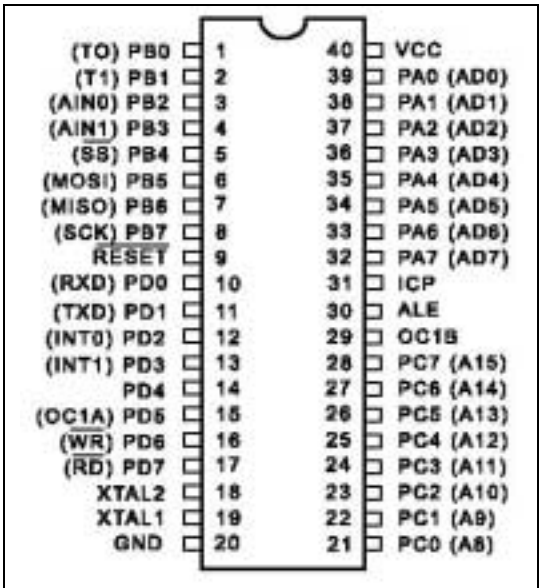


Fig. 5.2: Atmel AVR 90S8515 mC pin-out (PDIP package.)

Even though the proposed frequency of operation is 8 MHz, the AVR is smoothly running on a XTAL with a frequency of 11.0592 MHz (the datasheet [17] makes provisions for such a frequency). Apart from obviously pushing the performance (throughput) and speed of the mC to its limit, more practical reasons are provided in the SW description section of this chapter, for running the AVR at such high a frequency. A rough sketch of the mC external resources -distributed to the various subsystems- is shown in figure 5.3. The parenthesized names following some of the above port pins, e.g. PD2 (INT0), are indicative of the function mode to which they are internally set by the AVR, i.e. they are not functioning as conventional I/O pins. During this and the following chapter, the rationale behind the above-given resource assignment will become clear.

Subsystem	Resources Consumed
Input Interface	- PD2 (INT0) - PB1, PB2, PB3
Output Display	- Port A (i.e. PA7..0) - Port C (i.e. PC7..0)
UART Interface	- PD0 (RX), PD1 (TX)
BlueBridge[35]	- PD3 (INT1) - PB7 (SCK), PB6 (MISO), PB5 (MOSI), PB4 (/SS)
Various status LEDs (explained below)	- PB0 0: AVR offline, 1: AVR online - PD7 0: BlueBridge offline, 1: BlueBridge online - PD6 0: xUART not set, 1: xUART set - PD5, D4 Command mode

Fig. 5.3: AVR external resources assignment

5.2.2 The Input Interface

Let us now focus at the Input Interface. This subsystem's purpose is twofold:

1. to provide an abstraction of the underlying system complexity by simplifying all basic operations, e.g. search for Bluetooth devices in the vicinity (inquiry operation), set-up of a device so as to be visible by other devices, connection establishment between two Bluetooth modules etc.. On the other hand, in an effort not to "bury" the overall system functionality, this subsystem manages:

[35] To be further elaborated in chapter 6.

2. to give access to low-level features -an almost direct access to the HCI layer itself, e.g. change the settings of specific HCI commands, read the results of received HCI events etc..

With respect to the timing and AVR-resource constraints of the project, the optimum solution for the Input Interface subsystem has been a set 8 of push-buttons. Of course, having the AVR *directly* drive 8 buttons is out of the question: driving the buttons directly to 8 port pins and, then, polling these pins periodically for a change in the applied signals would waste valuable AVR resources and CPU throughput. On the other hand, attempting to use external INTs for all the buttons would be impossible since there are only two available; even by managing to use only the two of these INTs for the input subsystem, it would be prohibitory for the implementation of the rest of the system since there would be no external INTs left, which are crucial to the AVR for communicating with other HW parts (which indeed is the case, as seen in chapter 6).

The best implementation possible is the *multiplexing* of the button inputs through an encoder chip. Such a chip is **74F148**, an active-low 8-in-3 encoder[36] of the TTL family [18]. This is the reason 8 buttons (instead of 7 or 9) are used. Using, for instance, 10 buttons and a 16-in-4 encoder would do no harm but 8 buttons have proven to be sufficient in number for the design and also no resources are left unused. By using the 8-in-3 encoder chip, only 3 multiplexed signals need to be driven to the AVR. Yet, in order to avoid *periodic polling* of these 3 signals, an AND gate must be used whose inputs are the 8 signals generated by the 8 buttons and whose output is driven to one of the AVR external INTs (say INT0). All buttons, when not pressed, output a high-level signal, since the encoder uses active-low logic[37]. In this way, every time a button is pressed, it forces a low-level signal to the input of the AND gate, which -in turn- outputs '0', setting the INT0 pin low. At this point, the INT0 Interrupt Service Routine (hereon ISR) is executed and the 3-bit signal, which is continuously directed to the AVR, is polled once; the AVR decodes the 3-bit value, anticipates which button has been pressed and acts accordingly (specific button operations will be presented in the SW description). To make things somewhat simpler, from the 74F148 datasheet, it can be observed that apart from the 3-bit encoded output, two additional 1-bit outputs are supplied, one becoming high when any of the inputs is low (i.e. button pressed) and one

[36] The 74F148 chip is actually a 8-in-3 priority encoder ("F" stands for fast TTL). Both the "priority" and "fast TTL" features of the chip are of no concern to this project and the selection was based solely on availability at the time. The 74LS148 chip may be used as well.

[37] Active-low logic: the logic according to which a signal is considered active when becoming low. The opposite is true for the active-high logic.

which is the inversion of the first one. Should the second of those extra outputs be used (called "Group Signal Output", /GS), there is no need for an additional 8-input (i.e. multiple dual) AND gate.

Apart from that, *pull-up resistors* need to be used for all buttons in order to avoid the possibility of "floating" signals in the inputs of the encoder and -in effect- in the input pins of the AVR. This would cause the system to be unpredictably unstable by untimely detecting interrupts (in the INT0 pin) and by "reading" incorrect logical values in the 3-bit input.

A final yet substantial touch to the Input Interface subsystem is a *debouncing circuit* for the interrupt line. During development time, the well-known mechanical problems (such as glitches and sparks) caused due to button bouncing, have drawn added instability to the system, e.g. on a single key press, a few dozen interrupts were activated. The need to produce single and clear pulses to the various AVR inputs (and especially the interrupt line) has resulted in adding a debounce circuit [19, 20],. Apparently, the most sensitive and crucial signal needing debouncing is the interrupt line (INT0) so this circuit can be added serially between the /GS output of the encoder and the AVR interrupt input. The 3 encoded signals can also be debounced but there is practically no difference in the behavior of the overall circuit behavior since, no matter how many bounces these 3 signal do on each key press, sampling of their value is done only one (and late enough in time), when the ISR is activated. Because of the insertion of the inverter gate, the only configuration needed is to set the INT0 sense control to positive-edge triggering (instead of negative-edge). The final form of the Input Interface subsystem with the "/EO" output, the pull-up circuitry and the debouncing circuitry is the one depicted in figure 5.4.

The inverter, with the Schmitt-trigger input shown, is the one out of the hex-pack chip used: **74LS14**. This is a chip containing six independent inverters with input hysterisis [21], thus allowing both the INT0 line and the 3-bit encoded line to use a single chip -if so desired. Bringing back the table of resources of figure 5.3, the ones reserved for the Input Interface can now be explained: PD2, i.e. INT0, corresponds to the interrupt line triggered by the /GS output signal of the encoder and PB3, PB2, PB1 are used for the 3-bit encoded signal resulting from the 8 buttons.

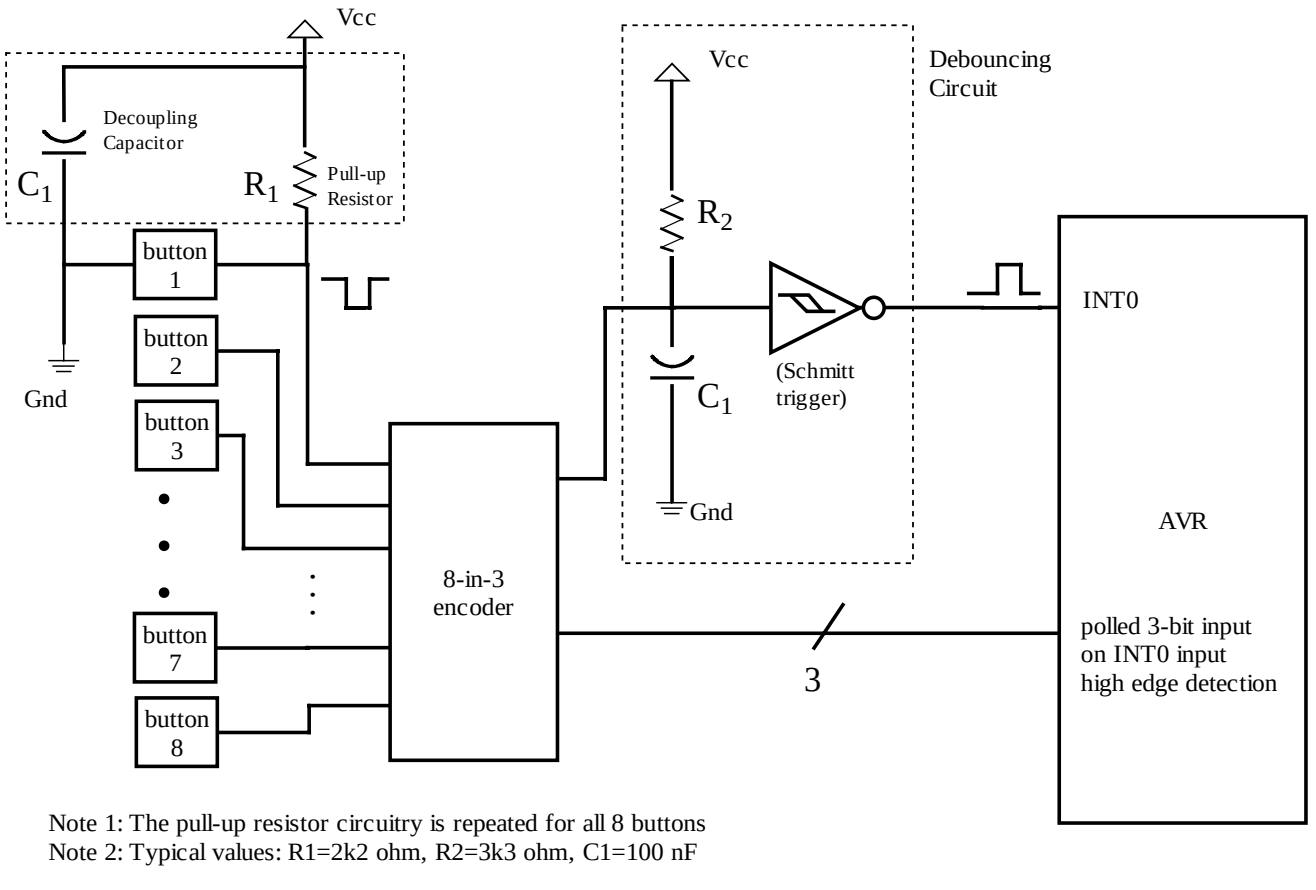


Fig. 5.4: Input Interface overall diagram

5.2.3 The Output Display

The operation of the above-explained buttons is gravely related to the Output Display. Behind the shiny name, this subsystem comprises (in parallel, see table of figure 5.3) of two ports of the AVR -port A and port C- each of which is responsible for driving 8 separate LEDs. Both rows of LEDs have multiple purposes displaying various pieces of information. More specifically, port A has the following tasks:

1. it displays the sequence number of the currently selected HCI command. This command is one of many implemented commands that can be selected for issue to the HC of the Bluetooth module (explained later on). Apart from HCI command indexing, it also provides some indexing of secondary functions of the system like the current baud rate for the external UART[38] etc., and
2. it displays additional details when an error in transmission or reception of an HCI_PDU has occurred (i.e. it displays the OpCode of the erroneous packet).

[38] The external UART is concerned with the BlueBridge application and is to be further elaborated in chapter 6.

Port C, on the other hand, is occupied with displaying various error or status messages during system operation. Details about both port A and port C functionality and a complete list of messages / indications will be given later on. A diagram of the whole Output Display is depicted in figure 5.5.

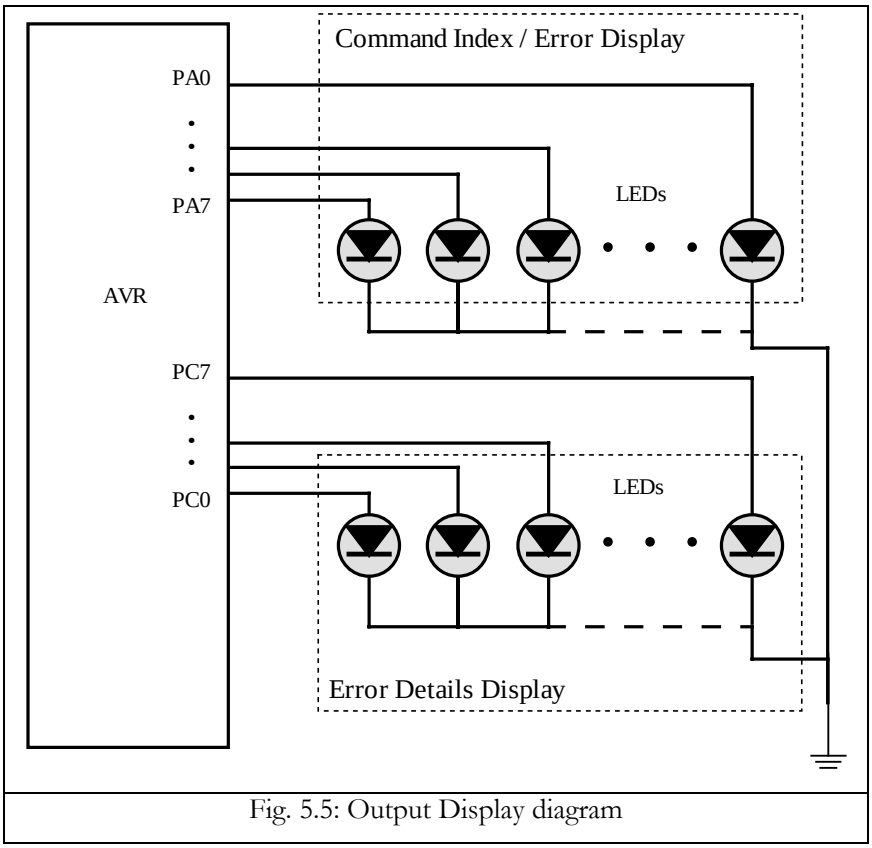


Fig. 5.5: Output Display diagram

It is obvious that this is a *waste* of resources, one way or the other. For instance, a 4-in-16 decoder could have been used (probably in combination with some transistors for boosting the low output current) giving the same results with only 4 reserved pins instead of 16. This is indeed true but the final decision was taken based on limitations in the development time. This subsystem is the one with the least effort spent on and has actually remained the same from the very beginning of this project; most time and effort has been directed to the rest subsystems with little attention[39] being paid at "face lifting" the way that events appear to the external user due to the tough time schedule. However, the 2x8 LED setup has proven to be sufficient for unveiling the various functions and states of operation of the system.

[39] An attempt has actually been made to use a 256x256 LCD screen by Seiko but we had a hard time finding almost any documentation or datasheets. Also, setting up the LCD screen to work properly would be a small project on its own account! [22]

More importantly, we have always kept in mind that the current Output Display subsystem is clearly a *temporary* one. The careful reservation of resources (in HW) and the flexibility of the host SW structure offer great (re)design flexibility which means that the subsystem now active can be removed *very easily* and substituted with a more sophisticated and probably less resource-consuming one, in a future version.

5.2.4 The UART Interface

The one subsystem yet not presented is the UART Interface. The reason is that this subsystem has a supplemental role to the whole design; it is an internal I/F between the host and the Bluetooth module. Over this I/F the HCI layer is substantiated and HCI traffic is exchanged in both directions. Apart from changing its speed with a vendor-specific HCI command (which is sent over to the Bluetooth module and informs it of the UART baud rate change), the external user can affect the UART Interface in no other way.

In figure 5.1 it can be seen that the UART Interface is not fully independent from the host; instead, one part seems to independently rest on the BlueApplE board while the remaining seems to be "claimed" by the AVR. The reason for this segmentation is that indeed one important part of the UART Interface is substantiated by the mC; this is the UART "back-end", so to speak. The UART HW parts -the transmit and receive registers, the control register (UCR) holding the UART operation settings such as the baud rate, the interrupts etc. and the status register (USR) holding the various UART flags such as data-register-empty, overrun, framing-error flag etc.- all reside inside the AVR. The mC is responsible for taking care of all the low-level tasks of the UART, thus providing a seamless operation.

The remaining part of the UART Interface is -by analogy- the "front-end"; this part takes up the task to formulate the UART signals so that communication with the Bluetooth module is possible. This is so because the Bluetooth module uses -as mentioned before- a fully-featured RS-232 protocol meaning that not only flow control signals are supported but also different voltage levels are applied than the plain-UART voltage levels of the AVR[40]. Obviously, a line driver must be used to convert the UART voltage levels to their equivalent RS-232 ones. For this purpose a **ADM232** (or **ICL232**) chip is

[40] The UART operates in the two voltage levels: [0V] – [5V], while the RS-232 operates in the two inverted voltage levels: [+12V] – [-12V].

utilized[41]. The HW part of the AVR enabling the UART operation combined with the externally connected ADM232 chip which makes the BlueApplE "HW-compatible" with the Bluetooth module, form the UART Interface subsystem. Its structure is depicted in figure 5.6. Referring to the table of figure 5.3, the pins reserved for the UART Interface, are PD0 and PD1, which are set by the AVR as RX (receive) and TX (transmit) lines, respectively.

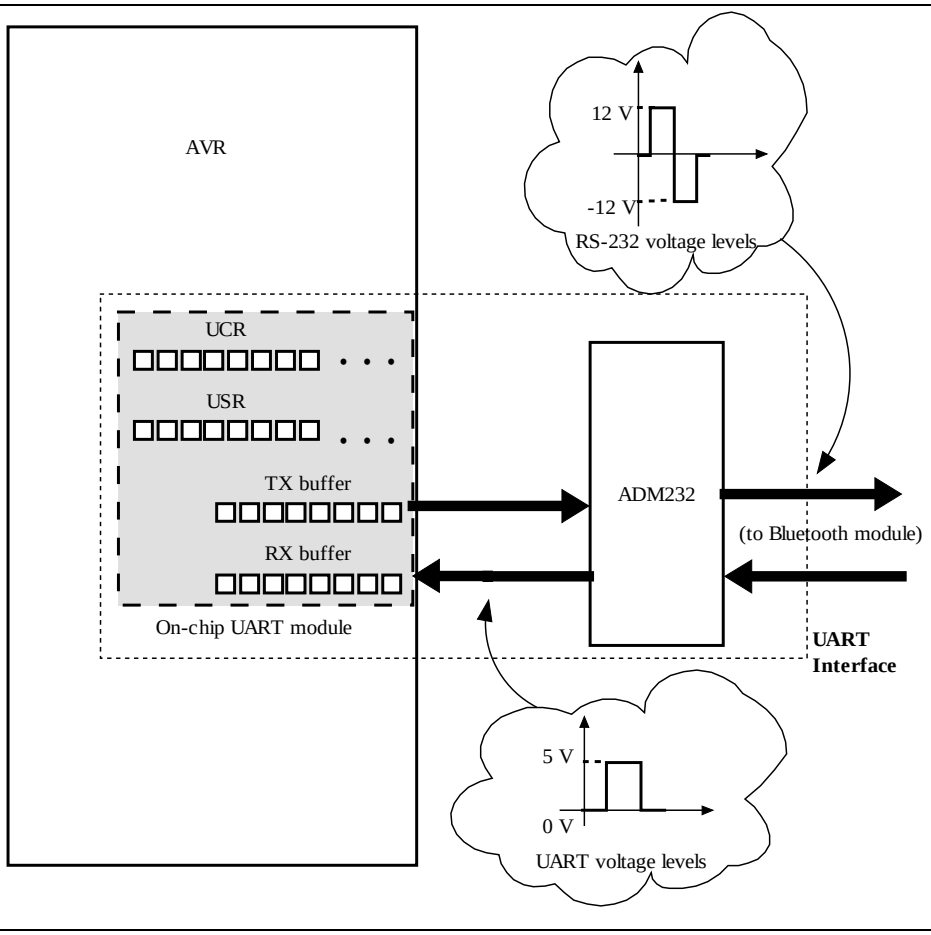


Fig. 5.6: UART Interface diagram

With the description of this subsystem, the HW walkthrough of the BlueApplE can be safely considered finished. The following sections are concerned with the various aspects of the SW design, a task far more demanding than the one just finished. However, this part of the document will also be more exciting since it is going to help put the pieces of the puzzle together and -thus- reveal the "greater picture", after all.

[41] For a more detailed view of the external circuitry of the ADM232 (or ICL232) chip, refer to [23].

5.3 Software Specifics

The development of the SW has been by far the most challenging aspect of this thesis. There has been a persistent effort to respect the features mentioned in the beginning (low power, low cost) while -at the same time- providing a portable, flexible, easily expandable and fully customizable design. In pursuit of such attributes, SW development has been modular in the sense that various modules have been built with explicit as well as implicit I/Fs among them; each module performs very specific tasks. In the following subsections, a thorough investigation of each module will be performed addressing issues of structure, functionality and resource reservation.

Before focusing on these modules, though, let us display the SW framework. A rigorous flow chart of the overall Host operation is depicted in figure 5.7. After power-up and proper configuration, the BlueApplE system enters an idle state waiting for events to happen. Such events are effectively triggered by a number of interrupts either external or internal ones. An external INT can be caused either by a local command *-block 1-* issued by a user through the Input Interface (examined previously) or by some incoming HCI event packet *-block 2-* from the local HC (residing on the Bluetooth Module) through the HCI transport (UART). In the former case, the system executes the corresponding ISR where it determines the type of order issued by the external user. Then, a specific action can be taken concerning either the Bluetooth Module under control *-block 3-* (HCI commands) or the Host itself *-block 4-* (internal / system operations)[42]. If an incoming HCI event packet has caused the INT, then proper reception and decoding of the incoming HCI packet takes place, followed by actions depending on the data included. Such actions may involve simple adjustment of the Host settings (system registers etc.), update of information regarding running connections, local or remote Bluetooth devices in the vicinity etc.. They may also involve the **automatic** issue of HCI commands to the local[43] Bluetooth Module, hiding various underlying HCI operations thus reducing system complexity and making system manipulation by the user more carefree. Internal INTs *-block 5* - have no impact away from the BlueApplE (i.e. on the Bluetooth Module). The ISRs triggered by them are solely bound to system functions with the task of providing smooth operation and are normally not controlled by the external user.

---

[42] All types of supported commands (HCI and internal ones) will be discussed next.

[43] The terms "local" and "remote" will be amply used hereon. "Local" refers to all parts of the BlueApplE and Bluetooth Module under examination while "remote" refers to similar systems which are located far from the *local* one. Simply put, *remote* systems are thought of as other Bluetooth-enabled devices in the vicinity of the *local* system which constitute potential members of a piconet initiated by the *local* system.

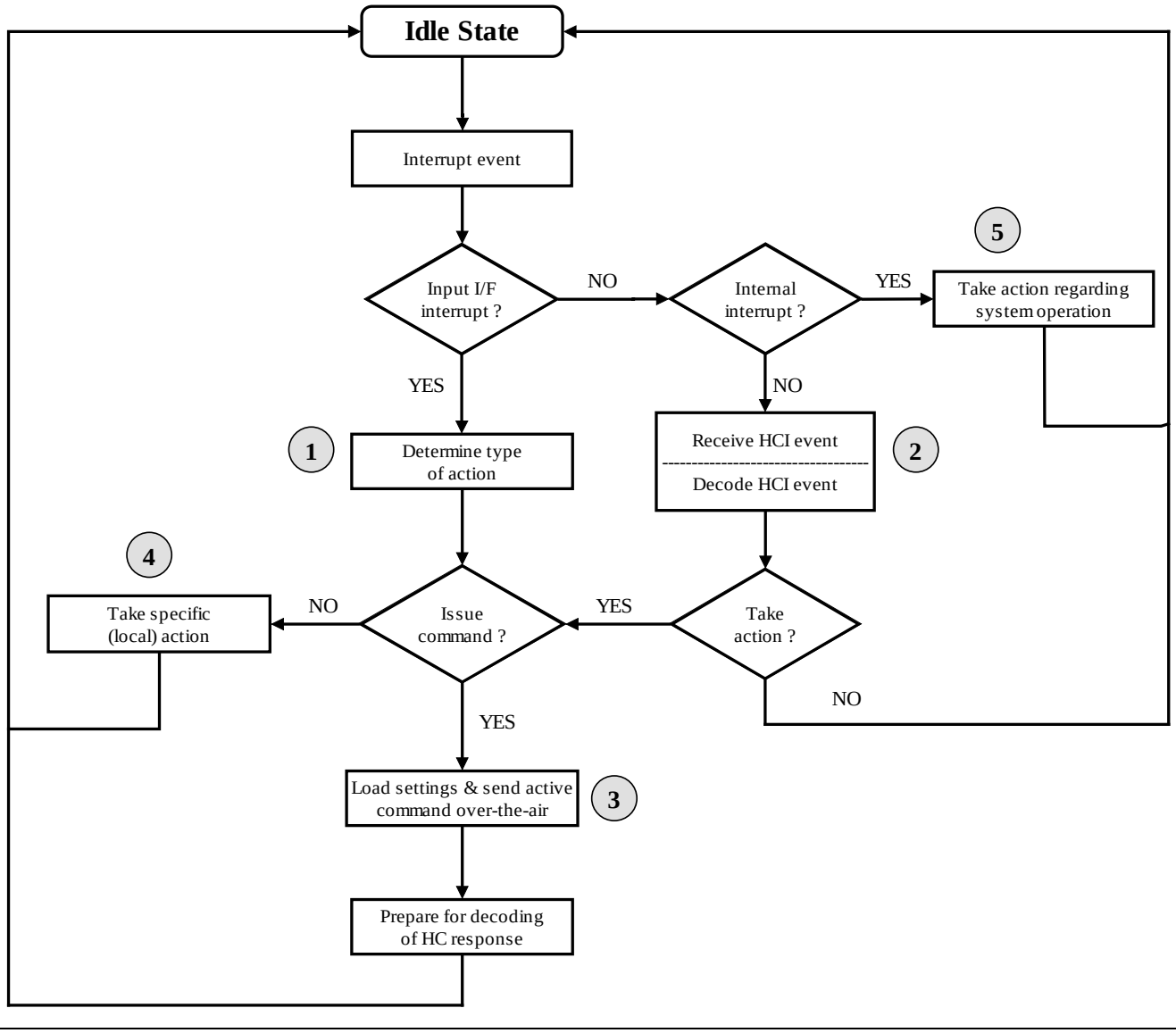


Fig. 5.7: Flow chart of Host state machine

More insight on the interrupt functionality and the various system modules is given in the following subsections. One last but significant point regarding the rest of the chapter is that all SW design was based on the Bluetooth Specification v1.1, the most recent version at the time. On the other hand, the "Bluetooth Applications & Training Tool Kit" used, has been qualified according to the Bluetooth Specification v1.0 (+ Errata). Yet, no problem has been encountered so far and -ergo- the code produced is compliant with Specification v1.1. Apart from that, great effort has been put at making the code non-device-specific, in terms of the Host and the Bluetooth Module. The obvious advantage is that in a future version of the project, a different Host device and also a different Bluetooth device can

be used with minor modifications only concerning some Host-specific internal configurations (e.g. in reserved-word terminology).

5.3.1 HCI commands

As discussed in chapter 3, a large number of commands is supported over the HCI layer. The HCI portion of the BT spec. is by far the largest one -nearly 300 pages. This should come as no surprise since the capabilities of the HCI define what can be accomplished with the Bluetooth technology. Since HCI defines the set of functions of a Bluetooth module that are accessible to the Host and its applications, HCI is the gatekeeper of the services that this module can provide to its users. Any feature of the module that is not exposed by the HCI, limits the functionality of the module.

Keeping the above in mind, a great deal of time has been spent at implementing the various HCI commands. Like all bit-fields defined in the BT spec., HCI command packets are structured in a Little-Endian manner; transmission takes place with the Least Significant Byte sent over the air first. However, the BT spec. has appeared somewhat "taciturn" in explaining how the various HCI packets are built -a controversial matter indeed during development time. For this reason a brief but insightful tutorial on how to form an HCI packet is included in Appendix C, since, explaining all the process here would only complicate things. Suffice to say that each formulated HCI command packet consists of at least 4 bytes: 1 byte for the packet indicator as seen in fig. 4.6, 2 bytes for the command OpCode and one byte for the Parameter Total Length being equal to '0x00' as no further parameter bytes follow, e.g. *Reset* command: '0x01' '0x03' '0x0C' '0x00'.

The Bluetooth Specification v1.1 defines a total of **95** HCI commands which are separated in different groups depending on their functionality (see chapter 3, OGF). Timing and Host-device constraints[44] have made implementation and full support of all 95 HCI commands impossible; only 27 (about 1/3) of them have been included in the final version of the specific SW module. Yet, the key observation here is that only a small portion of those 95 HCI commands is directly responsible for enabling the *most significant* operations of the Bluetooth technology, such as the search for devices in the vicinity (*Inquiry* command), the activation of a device in order to be visible by others (*Page_Scan_Enable* command) and the creation of a connection between two devices (*Create_Connection* command), to name a few. This

[44] For a more detailed discussion on the various design limitations stemming from the technology used, see chapter 8.

does not diminish -however- the importance of the remaining HCI commands which are responsible for a plethora of substantial operations occupied with providing correct, secure and untethered communication between devices. Having included in the implementation -among others- the most crucial HCI commands (which are also the most difficult to handle), a total of 27 HCI commands has proven to be more than adequate for unleashing diverse features of the Bluetooth technology. Even the BT spec. suggests that in cases where embedded systems are involved (such as the one at hand), one needs not implement the whole HCI layer, so this thesis means no violation of any kind to the Bluetooth protocol stack. What is more, the way these 27 commands have been implemented inside the Host, makes the addition of more of them an extremely trivial and typical matter, as explained below. In figure 5.8, a complete list of BlueApplE-supported HCI commands is cited, organized by group (OGF) and accompanied by brief explanations. For additional details on their purpose and functionality please refer to the Bluetooth Specification [4]. Only one observation shall be made here: the Ericsson-Specific command responsible for altering the UART baud rate has been found in the ROK chip manual [14]. One can find there a complete list of supported baud rates ranging from 300 bps to 460.8 bps. On power-up or HW reset, the Bluetooth module communicates via the UART at 57.6 kbps so this is the baud rate at which the UART of the AVR is also set initially. Yet, the *maximum* baud rate supported by the AVR is 115.2 kbps; ergo, the Ericsson-specific command is statically written to change the UART baud rate of the Bluetooth module to only 115.2 kbps when issued. At the same time, the AVR obviously also sets its own UART baud rate to the same value.

HCI command name	Details
Link Control Commands (OGF: '0x01')	
Inquiry	Searches for active Bluetooth devices in the vicinity and returns relative information.
Create_Connection	Attempts to establish an ACL connection to another device based on the remote device's BD_ADDR. On success, a *Connection Handle* is assigned to the ACL link.
Disconnect	Terminates an existing ACL connection based on its Connection Handle.
Accept_Connection_Request	Accepts an incoming connection request (thru a preceding Create_Connection).
Remote_Name_Request	Obtains the user-friendly name of a remote device based on its BD_ADDR.
Host Controller & Baseband Commands (OGF: '0x03')	
Rst	Resets HC & LM. The local device enters stand-by mode; HC assumes default values.
Set_Event_Filter	Specifies different event filters (i.e. the Host receives only events that interest it)[45].

[45] The PIN code is used during authentication & pairing of Bluetooth devices (as seen in chapter 3, LMP). For more details on the subject see [4]: Part C.

Write_PIN_Type	Lets the HC know if the Host supports variable PIN codes (used in pairing) or not.
Read_PIN_Type	Reads whether the LM assumes that the Host supports variable PINs.
Change_Local_Name Read_Local_Name	Writes/Reads the user-friendly name of the local Bluetooth device.
Read_Connection_Accept_Timeout Write_Connection_Accept_Timeout	Reads/Writes the parameter holding the amount of elapsed time needed for a device to reject a (remote) connection request.
Read_Page_Timeout Write_Page_Timeout	Reads/Writes the parameter holding the amount of time for which the HC waits for a remote device to accept a locally issued connection request before considering it failed.
Read_Scan_Enable Write_Scan_Enable	Reads/Writes the parameter deciding whether a device will perform periodic inquiry and/or page scans so as to be visible by remote devices, or not.
Read_Authentication_Enable Write_Authentication_Enable	Reads/Writes the parameter deciding whether the local device requests authentication of the remote device at connection setup, or not.
Read_Encryption_Enable Write_Encryption_Enable	Reads/Writes the parameter deciding whether the local device requests encryption of the remote device at connection setup, or not. (Authentication must also be enabled).
Informational Parameters (OGF: '0x04')	
Read_Local_Version_Information	Reads local information: HCI ver., HCI revision number, LMP version, Manufacturer name, LMP subversion
Read_Local_Supported_Features	Requests a list of LMP-supported features.
Read_Buffer_Size	Reads the max size of the ACL & SCO packet *payload* sent from the Host to the HC.
Read_Country_Code	Reads the Country Code; it defines which part of the ISM 2.4 GHz band is used.
Read_BD_ADDR	Reads the BD_ADDR of the local device.
Ericsson-Specific HCI Commands (OGF: '0x3F')	
Ericsson_Set_Uart_BR	Changes the baud rate of the UART (HCI transport). Default value: 57.6 kbps.

Fig. 5.8: Supported HCI commands

5.3.1.1 Internal organization

The trick in making HCI command management in the Host "cushy" resides in the structure developed. Every byte sequence of limited length representing a specific HCI command (e.g. Reset command: '0x01' '0x03' '0x0C' '0x00') is stored in an AVR internal, volatile memory. This is a (512 x 1) bytes SRAM directly mapped inside the AVR. Since it is a volatile memory, on every power up, all the supported HCI commands must be written in it. Each entry is sequentially written after the previous one, e.g. as seen in the table of the above figure 5.8, the Create_Connection command bytes are written *right after* the Inquiry command bytes. Every time a new command is added, its starting address in SRAM is assigned to a **label**. To make the code flexible and non-device-specific, each label (holding a new command address) is created based on the label holding the address of the previous command plus an

offset equal to the byte length of the previous command. For instance, given that the Inquiry command takes 8 bytes in SRAM and is written first in the SRAM (SRAM_START label), the next command, Create_Connection, will be written 8 memory slots after the SRAM beginning address:

```
.equ Inquiry = SRAM_START   ; (LENGTH = 8)
.equ Create_Connection = Inquiry + 8   ; (LENGTH = 16)
.equ Disconnect = Create_Connection + 16   ; (LENGTH = 6)
.equ . . .
```

The ".equ" is a reserved directive in AVR assembly for creating labels. Observe that in the way described, no memory address numbers appear and adding a new command, say FOO of length 4, between the 1st and 2nd one would require the following action:

```
.equ Inquiry = SRAM_START   ; (LENGTH = 8)
.equ FOO = Inquiry + 8   ; (LENGTH = 4)
.equ Create_Connection = FOO + 4   ; (LENGTH = 16)
.equ Disconnect = Create_Connection + 16   ; (LENGTH = 6)
.equ . . .
```

The only changes needed are highlighted in gray. Of course, apart from the addressing addition, the actual new command bytes must be written in SRAM in the proper position.

5.3.1.2 Management through Indexing

While this mechanism has helped to effectively manage HCI command additions/modifications, another one has been sought for making access to all these HCI commands a rather simple issue. For this purpose, another type of memory available in the AVR -the program memory- has been utilized in this SW module. The program memory is a non-volatile Flash memory of size 2K x 2 bytes (i.e. 4096 bytes). The AVR provides the functionality of *loading data from the program memory* during normal operation. This gives the opportunity to create a look-up table (hereon LUT) in which the starting address (in SRAM) of every implemented HCI command will be stored. Since SRAM addresses are 2 bytes long (0xXXXX) i.e. the same size as each slot of the Flash, one HCI starting address can be written per Flash slot. In so doing, we manage to create a list of consecutive SRAM addresses (a LUT) inside the Flash; this list can be accessed in a most typical way, by using standard, well-defined assembly

commands for reading a specific address. The LUT created in this way, in virtue renders a list of pointers to the SRAM-located commands. Since the order in which commands are located in the SRAM and in which their starting addresses are located in the Flash is the same and, given that each such address reserves one Flash slot, a one-on-one relationship is created. Then, each HCI command is assigned a serial number -referred to as a *command index* (see below)- ranging from 0 to 26 (or 1 to 27); this number defines how far from the beginning of the LUT we must look for the starting address of the command owing this number. For instance, with reference to the table of figure 5.8, the Write_Scan_Enable command will be assigned the number 16. Then, the starting address (in SRAM) of this command can be found in the LUT (in Flash) by advancing 16 times from the start of the LUT. By reading the specific entry we will come up with the proper address. Of course, when a new HCI command is added to the Host, the additional task of informing the LUT is required, which is trivial given that all the work has been done without explicitly "remembering" any address number etc.. The formed LUT looks like this:

```
.dw Inquiry
.dw Create_Connection
.dw Disconnect
.dw . . .
```

The ".dw" directive stands for "define word" and is used for directly writing a word (here: the starting address of a command) in Flash. Also, the "Inquiry", "Create_Connection" etc. strings are the labels defined previously (during command write in SRAM). As in the above case, an example of adding a new command, say FOO, would have the following effect in the LUT:

```
.dw Inquiry
.dw FOO
.dw Create_Connection
.dw Disconnect
.dw . . .
```

Summing up these two tricks, the effort needed to insert/modify a new HCI command is limited to the grey highlights in the above code segments making it a trivial task. By numbering the various HCI commands, a user-friendly I/F is offered to the external user (that is, with embedded systems in mind).

This representation is very handy for both the external user and the various internal (system) operations. Through the use of the Input Interface (supported operations examined later on) or internally, BlueApplE can rapidly issue any HCI command desired. Figure 5.9 depicts the SRAM contents, as previously described, and also the Flash-enabled pointer system and its interaction with the SRAM[46].

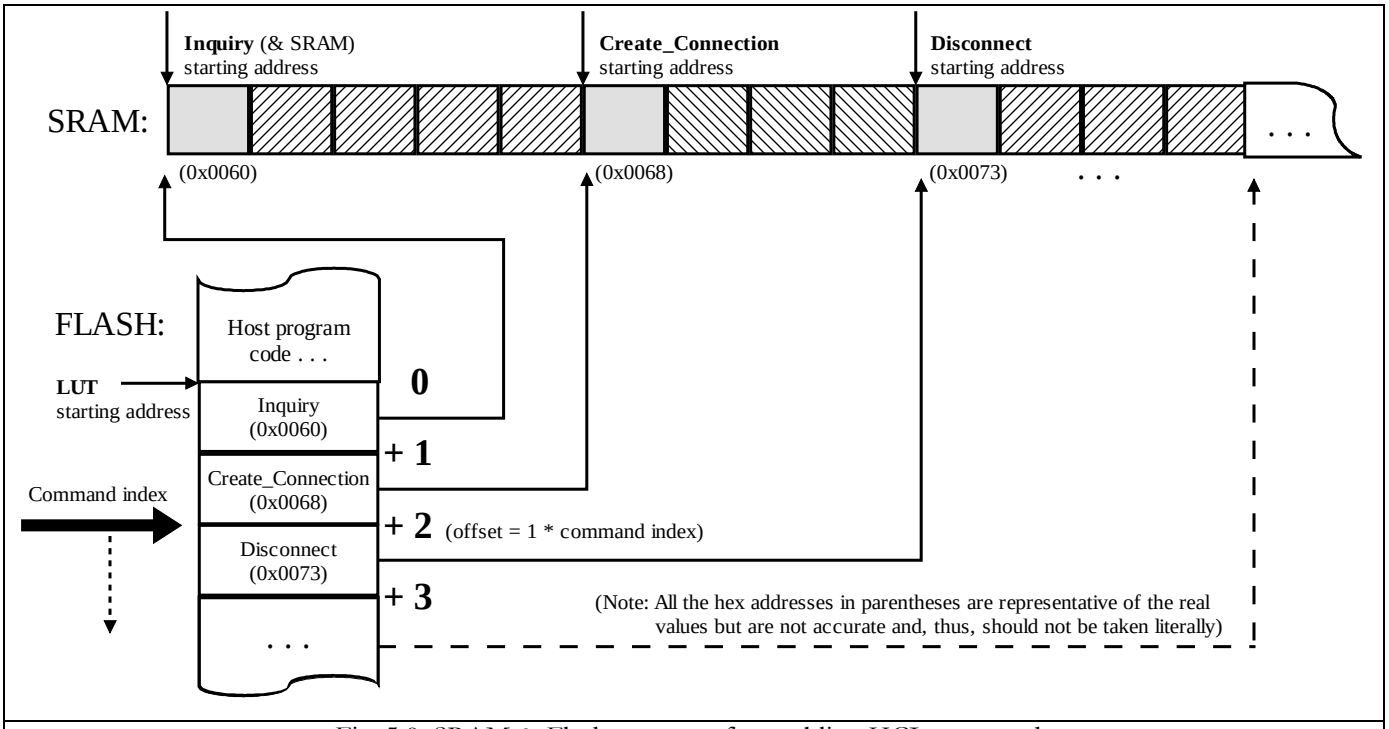


Fig. 5.9: SRAM & Flash structure for enabling HCI commands

Except for managing the structure and access method of the various HCI commands, it is expected from this SW module to be also responsible for properly setting the various dynamic byte-fields of the HCI commands and for issuing them over the HCI transport (UART). For instance, the Write_Scan_Enable command has the following fields: '0x01' '0x1A' '0x0C' '0x01' '0x03'. The first byte is the HCI packet indicator (command packet), the second and third constitute the command OpCode, the fourth byte is the number of parameters and the fifth one is the parameter itself, i.e. the mode of page scan to select from. Of all the above fields only the last one is dynamic and can be set to '0x03' (as in the above case) meaning that the Bluetooth Module is open to both inquiries and pages by other

[46] Whereas the specific mC used, AT90S8515, has the ability to *load* data from the program memory in normal operation, it cannot *store* data. Such a feature would provide a more uniform solution than the one just described (e.g. by storing all HCI commands AND the LUT in Flash) and would also save precious space in the SRAM which is far smaller (512 bytes) and faster than the Flash (4 Kbytes). Limited SRAM has indeed proven to be a drawback in this thesis and a limiting factor, too.

devices. All those dynamic (parameter) byte-fields may be either externally (by a user) or internally fed (through data acquired via previously issued Inquiry commands, Create_Connection commands etc.) or even statically written in SRAM for the purposes of this thesis (as is the case of the Write_Scan_Enable). It is the job of this module to detect missing or erroneous bytes in the packets to be transmitted, in which case command issue is cancelled and explanatory error messages are generated.

In the next subsection, the way decoding of received HCI events is performed inside the Host will be explained. This process is gravely related with the HCI command structure explained above.

5.3.2 HCI events

HCI commands are not the only ones to be kept in SRAM. Byte patterns of HCI events are also stored in it. For several HCI commands, information related to their status and execution results is carried by two special events: the *Command_Status_Event* and the *Command_Complete_Event*. The former typically is sent immediately after a command is received by the HC to indicate the status of the command such as "command pending execution", "command not understood" and so on. This provides a sort of acknowledgement of the command along with an indication of its processing status. The latter is used to indicate the completion of execution of a command and to return related parameters, including whether or not the requested command has executed successfully.

Apart from those basic HCI events corresponding to the majority of HCI commands, many more exist for specific HCI commands (in the BT spec. 32 different events are defined). It is obvious that, in order to decode the incoming events correctly and take further actions depending on the information they bear, specific measures must be taken by the Host in **two steps** of its operation. Before advancing to these steps, the rationale behind the HCI command-event pattern must be explained. Generally speaking[47], there are three scenarios:

1. the local Host issues a command concerning the local HC, e.g. Write_Scan_Enable. The HC takes specific actions and then transmits back to the Host a Command_Complete event. This is the most "popular" event –a generic event returned for **all** locally issued HCI commands.
2. the local Host issues a command concerning a remote HC, e.g. Remote_Name_Request. In such cases, the remote Host needs not be disturbed. Since such commands are not carried out locally, on issue, the

[47] Sporadically, differentiations in these cases exist but will be not investigated further for the sake of simplicity. For more details on the subject see [4]: Part H:1.

local Host receives a Command_Status event "reassuring" it that the command has been sent over the air. The remote HC receives the command, executes it and returns an event to the local HC with the results, without intervention from the remote Host since such commands are usually requesting low-level information residing in the LMP which is directly controlled by the HC. Finally, the remote HC further pushes the event back to the local HC and, in turn, to its Host. In this case, command-specific events are returned, e.g. for the Remote_Name_Request command, the Remote_Name_Request_Complete event is returned in response.

3. the local Host issues a command concerning a remote Host since it requires some *decision making* on the part of the remote device, e.g. Create_Connection. The packet traffic is the same as in case (2) with the difference that in this case the remote HC actually forwards the command to its own Host for execution.

Graphical examples of the above cases are depicted in figure 5.10.

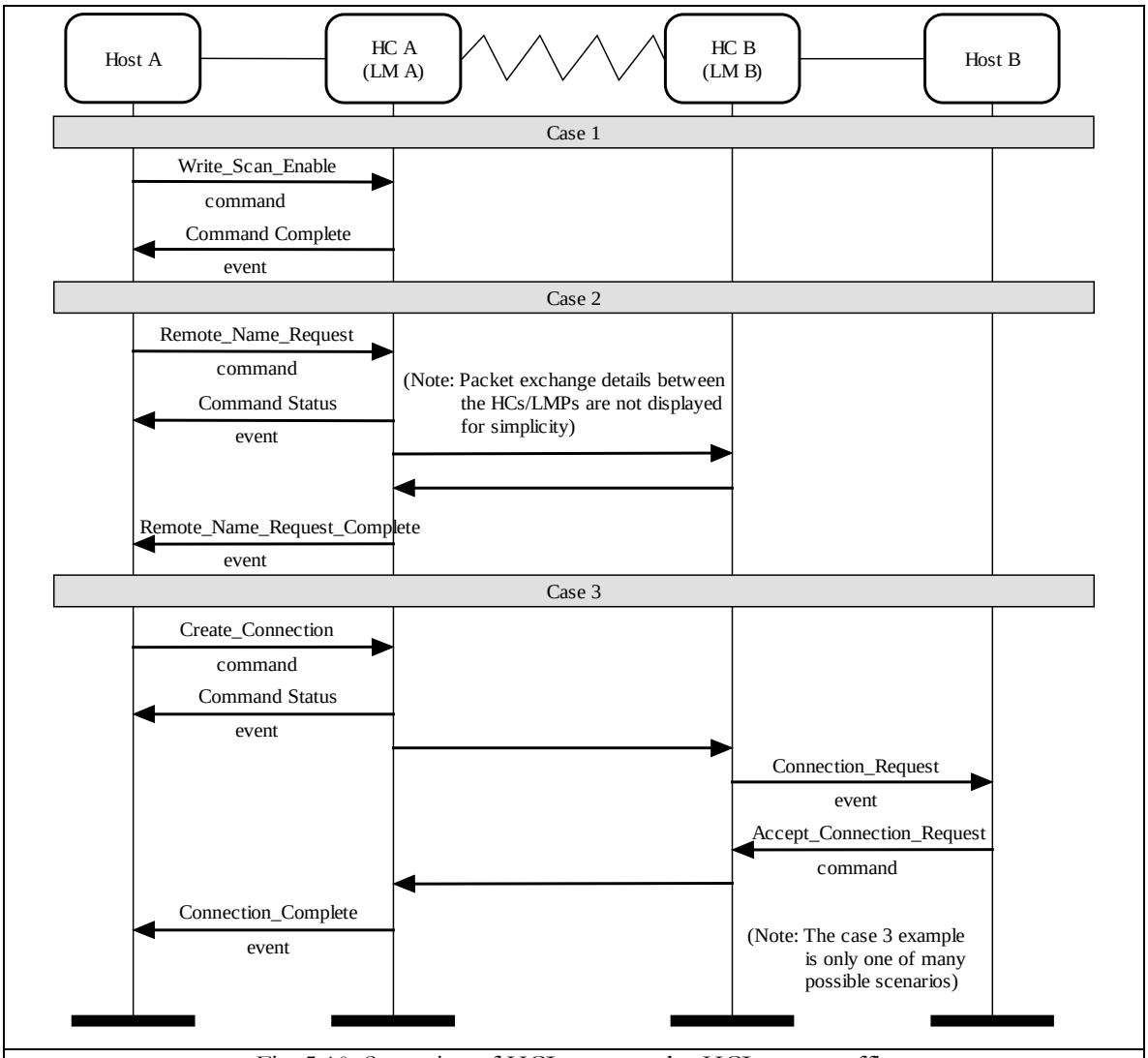


Fig. 5.10: Scenarios of HCI command – HCI event traffic

In all three cases, the local Host always knows what number and what kind of packets to expect since it is the one to trigger them by issuing commands. In case (3), however, a remote Host is also involved and -what is more- does not a priori know what packets to expect; for instance, in the case of the local Host issuing a Create_Connection command, the remote Host receives a Connection_Request event for which -obviously- no prior notice has been given. Additionally, when ACL and/or SCO links have been established, ACL and SCO data traffic is also exchanged apart from control packets. The problem of "unexpected" incoming packets exists also in this case, for both the Hosts (local and remote) since none of them knows the exact time the other one will transmit a data packet. In any case, every incoming

HCI event (and data) packet has to be received by the Host and properly decoded. Based on the information provided during the decoding process, specific actions may need to be taken by each Host. Such actions may be: sending requested information to the Output Display, automatically generating and transmitting some HCI command, updating (Host) system registers and forwarding a data payload to upper layers (e.g. some application), to name a few.

5.3.2.1 Decoding preparation

The **first step** for smooth decoding of incoming events is to *prepare* the Host for them: just like the HCI command bytes (seen in the previous subsection), some HCI event bytes are also written in the SRAM. These byte-fields are serially written at the end of the last implemented HCI command and actually provide **masks** of HCI events rather than entire, specific HCI events. The reason for using masks instead of specific instances of events is that some fields of the events are *static*, such as the event code (distinguishing a Command_Status from a Command_Complete etc.), while the event parameter fields are a priori *unknown* and are, thus, initially left blank. For convenience, the format of the HCI event packet of fig. 3.16 is repeated in figure 5.11. More that one mask exists in SRAM; whereas the general event format is the same, the parameters vary gravely in type and size. So, one mask is kept for Command Complete events, one for Command Status events, one for command-specific events and some more masks discussed later on.

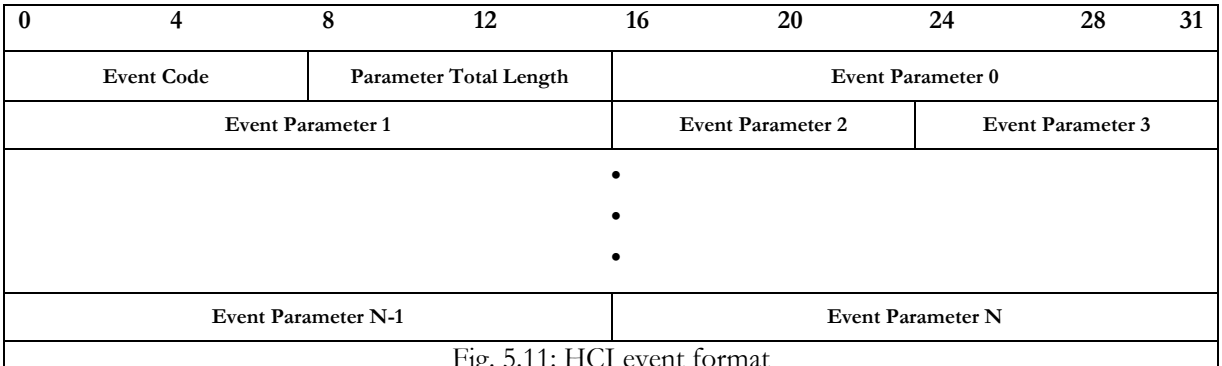


Fig. 5.11: HCI event format

When the Host issues a new command, it knows exactly[48] the number and type of HCI events expected in response, i.e. it knows the sequence (as seen in fig. 5.10) and format of the events (as seen in fig. 5.11) but obviously does not know the specific values of the parameter fields to be returned. However, this is sufficiently adequate a knowledge to prepare for decoding. To do so, *a second* LUT is formed inside the Flash, right after the previous LUT. In this case, 2 memory words (i.e. two slots) are reserved for each HCI command holding relative information of the event(s) expected: the **type of the event** (1 byte), the **parameter length** (2 bytes) and the **issuing-command OpCode** (2 bytes), which is most often included in the received parameter bytes. The type of the event is equal to '0x00' when a Command_Complete event is expected (for which the event type is statically set to '0x0E') and it is ignored during decoding preparation; in all other cases, this field varies and holds the event code of the expected command-specific event (e.g. '0x07' for the Remote_Name_Request event). Examples of how these bytes are actually used are mentioned later on.

On command issue, the above-mentioned blank event fields residing in the SRAM, are filled with the information regarding this specific command. The way the 4-byte-long event settings (in Flash) are matched to the currently issued command (in SRAM) is similar to the one used for matching Flash entries to command starting addresses: through the *command index* (as explained above). The only difference is that for each unit of the command index, two Flash entries are jumped instead of one. For instance, for the Read_Scan_Enable command with command index value equal to '15', we are advancing 30 words in the Flash before reading the 4 bytes needed for properly setting up the event decoding process for the specific command. This is graphically explained in figure 5.12.

[48] An exception lies with the Inquiry command case: depending on the number of devices existing in the vicinity of the inquiring device, a varied number (or size) of events is expected. This number is not a priori known to the inquiring device.

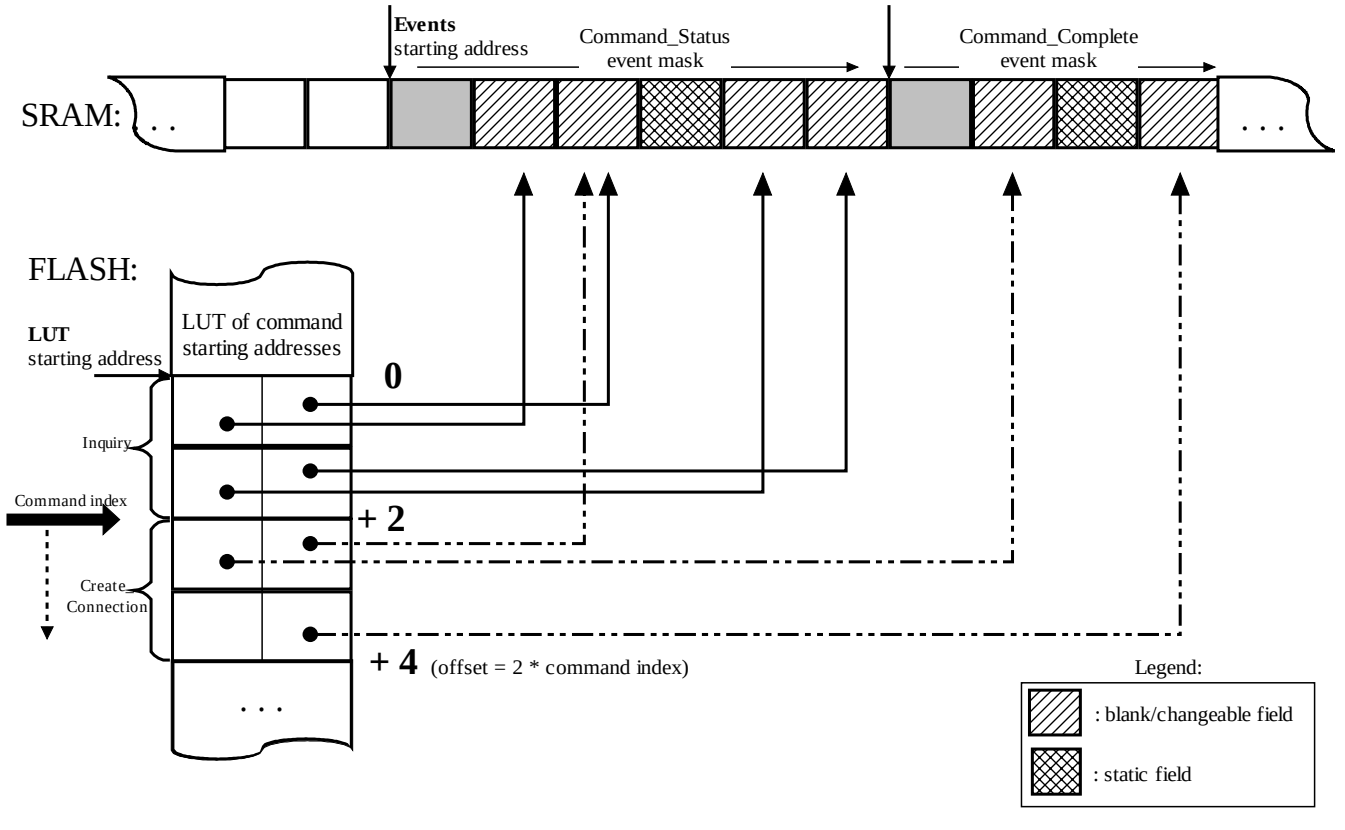


Fig. 5.12: HCI event decoding preparation

LUT contents are as follows:

```
;  (type  len   opcH  opcL)
.db 0x02, 0x0F, 0x01, 0x04   ; Inquiry
.db 0x03, 0x0B, 0x05, 0x04   ; Create_Connection
.db 0x05, 0x04, 0x06, 0x04   ; Disconnect
.db . . .
.db 0x00, 0x05, 0x19, 0x0C   ; Read_Scan_Enable
.db 0x00, 0x04, 0x1A, 0x0C   ; Write_Scan_Enable
.db . . .
```

The ".db" term is similar to ".dw" and stands for "define byte" in Flash. Four bytes are written: the *event type*, the *event parameter total length*, the issuing-command *OpCode* high byte and the issuing-command *OpCode* low byte, respectively. If the command issued is an Inquiry, the bytes of the above first line (in Flash) are written in specific slots of the SRAM. If, later on, a Create_Connection command is issued,

the same slots will be *overwritten* with the bytes of the second line etc..[49] In this way, the Host holds at any moment *valid* patterns of expected events. These patterns will be used during actual decoding for byte-to-byte comparison purposes (examined below).

An example of such an operation is given in figure 5.13. The command issued is a Read_Scan_Enable, which reads the value for the Scan_Enable parameter (from the local HC). The Scan_Enable parameter controls whether or not the Bluetooth device will periodically scan for page attempts and/or inquiry requests from other Bluetooth devices. The event returned is a Command_Complete one distinguished by the specific event code: '0x0E' so the mask used is the one reserved for Command_Complete events. Static return parameters of such an event are the issuing-command OpCode, the event Status and the Number Of HCI Command Packets (i.e. the Number of HCI command packets which are allowed to be sent to the Host Controller from the Host.). The only command-specific parameter is Scan_Enable. With the exception of this last parameter, all the previous ones can be "predicted" and can, thus, be written in the corresponding mask in SRAM, during command issue.

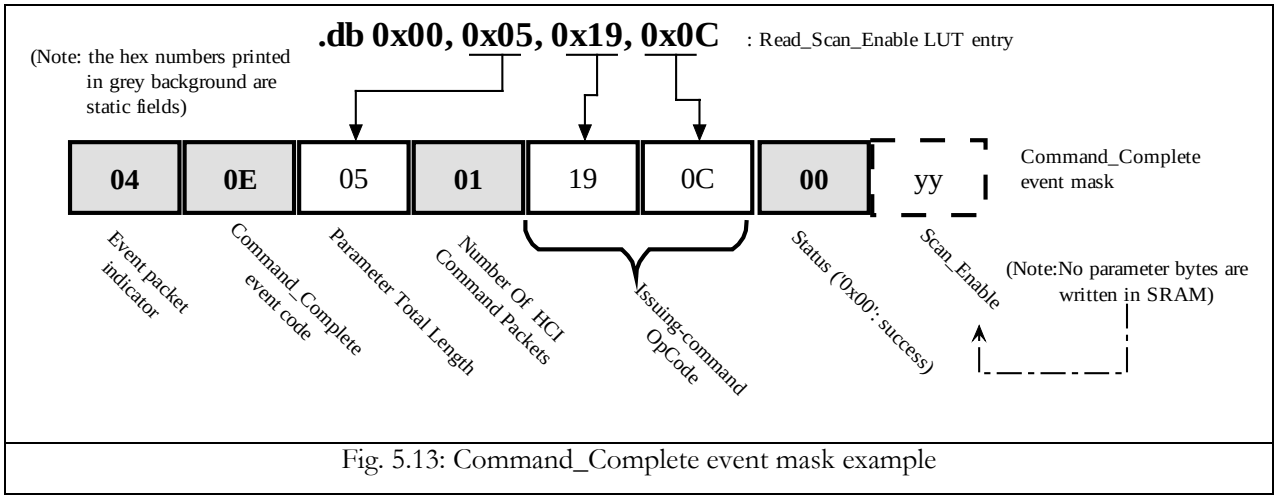


Fig. 5.13: Command_Complete event mask example

As seen in the figure, by reading the LUT in Flash for the Read_Scan_Enable command, the entry acquired is as follows:

```
.db 0x00, 0x05, 0x19, 0x0C   ; Read_Scan_Enable
```

[49] Actually, this decoding preparation algorithm is a bit more complex than described: depending on the event(s) expected, some of the above byte fields might be written in different event positions (in SRAM) or may not be written at all.

Since the event to be received is a Command_Complete one, the first byte ('0x00') is ignored, the second and third are written in the OpCode slots of the mask and the fourth in the Parameter length slot. The remaining static fields (Status and Number Of HCI Command Packets) are never changed in the mask and are initially written on AVR power up.

In the case of a command triggering a Command_Status event and a command-specific event, such as the Remote_Name_Request_Complete event for the Remote_Name_Request command, the various bytes of each LUT entry are used somewhat differently. In this case two masks are written: one for the Command_Status event and one for the command-specific event expected. The corresponding LUT entry in this case is as follows:

```
.db 0x07, 0xFF, 0x19, 0x04   ; Remote_Name_Request
```

The first byte is the type of the triggered command-specific event and will be written in a mask used for decoding such events; the same goes for the second byte which is the parameter length of this event. Only the OpCode bytes (third and fourth) are written in the mask used for the Command_Status event (the command-specific events never include the issuing-command OpCode in their parameters). This is depicted in figure 5.14.

It must be stressed that on all cases of received events, **no** parameter bytes are written in SRAM (e.g. the Scan_Enable parameter in the former case). The masks are used only for checking the overhead bytes of the various received events; the parameters are processed, checked and utilized *on the fly*.

Typically, the function just explained, is in the jurisdiction of the previous SW module (the one managing the HCI commands) since it is executed right after a command issue. The whole idea was presented here though, since it is directly concerned with HCI event decoding.

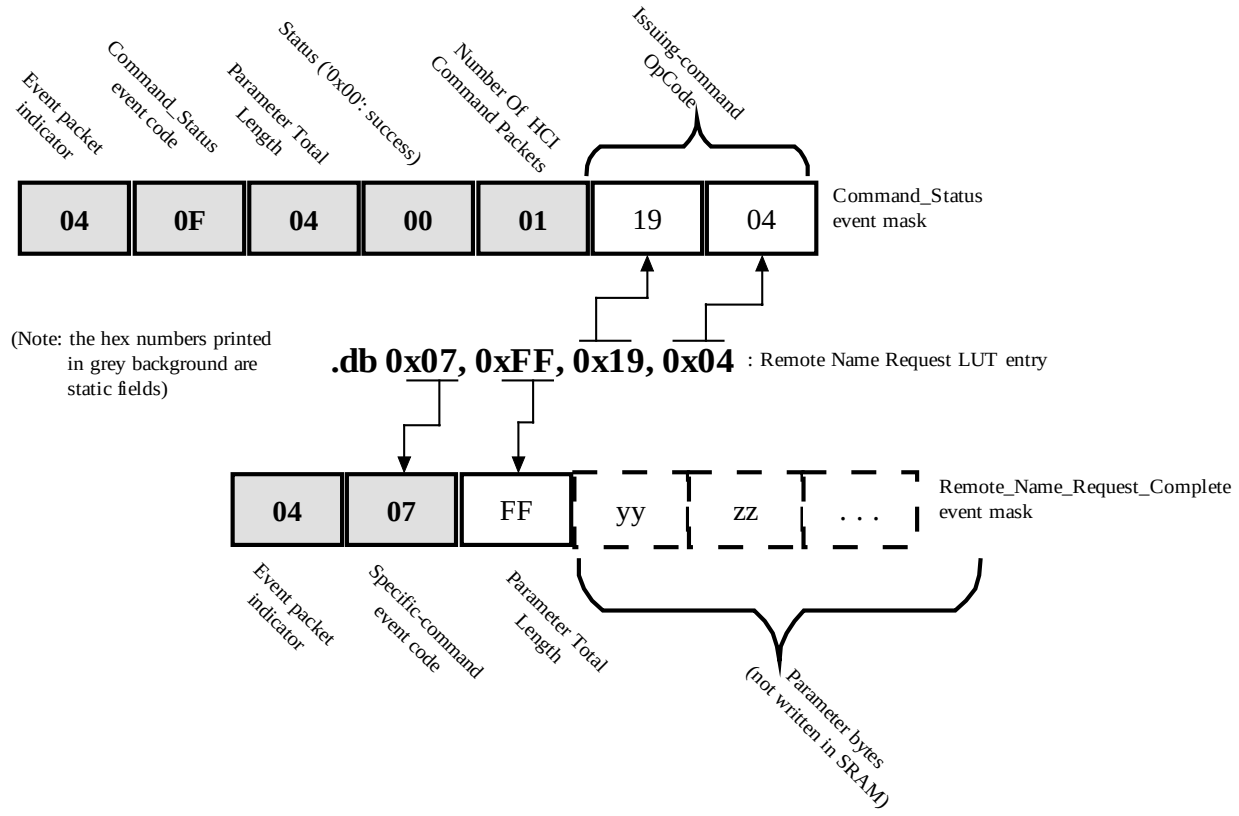


Fig. 5.14: Command-specific event mask example

5.3.2.2 Decoding process

Fatefully, the **second step** for event decoding is the decoding process itself which is activated when event (or data) packets finally begin to arrive. This decoding process is the core component of the currently described SW module and is based on a subroutine residing right on top of the UART I/F and waking only when it senses traffic on the channel (i.e. is the UART ISR). This subroutine "audits" all traffic coming from the (local) HC and tries to distinguish meaningful arrays of bytes. Of course, this scanning for packets is not random; the subroutine is properly informed -by the process explained above- which packets to expect and in what order. The incoming packets undergo comparisons with the previously-filled event masks (in SRAM). If all overhead bytes (such as codes, lengths, types etc.) check out, parameter data are examined. Checks are performed on those as well -whenever possible- and various system decisions are made.

Of course -as previously seen- some incoming packets cannot be "foretold" (ACL/SCO data packets and remotely-triggered commands). In such cases, the decoding routine has indeed no clue about any potential incoming packet but tries to distinguish the received packet as it arrives, all the same. Continuous and exhaustive checks are performed on the incoming packet to determine its type and validity; no assumptions are made on all incoming traffic whatsoever. If it does not much any existing pattern or appears to contain invalid / superfluous data, it is discarded, no further action is taken and explanatory error reports are generated (examined later on).

Great effort has been put at building a decoding scheme that is extremely space efficient and easily adaptable. It is a hybrid of two major decoding styles: a **binary tree** and a **buffer processing scheme**. The latter is the primary one which is enabled by the algorithm explained in the previous subsections: event masks are filled with valid data on command issue and incoming events are byte-size compared with these internal masks. This style can be used only for **predictable** events (i.e. events triggered by the local Host) which constitute the vast majority of HCI traffic, while at the same time provides a very fast means for decoding. Its key feature is its extreme adaptability since it can be easily expanded to support more types of events and more types of resulting actions in a direct, stack-up manner. On the other hand, the former style is used for **remotely-triggered (unpredictable)** events. While in the buffer case event patterns are dynamically created and matched with the actual events, in this case patterns cannot be built and decoding happens on a per byte reception basis. When the first unexpected byte arrives (packet indicator), the decoding engine tries to figure out what kind of packet it is. If this is an event packet, then the second byte determines the event type, the third the length of the event parameters etc.. If the first byte is indicative of an incoming ACL packet, handling of the following bytes will be different than above.

The above decoding styles are clearly separated and activated, based on some underlying structures responsible for turning them on and off. A complementary role to the Flash LUT (the one with the event data) plays a system register ("dcd_flags", i.e. decoding flags register) holding the sequence of all possible incoming packets. The flags of this register are depicted in figure 5.15. This register is checked whenever a new byte is received. The error flag is the first one to be checked. It is set when at least one erroneous byte has been encountered during reception. While set by a received byte, all following bytes will be damped until it clears. The mechanism behind error handling is examined in the next subsection.

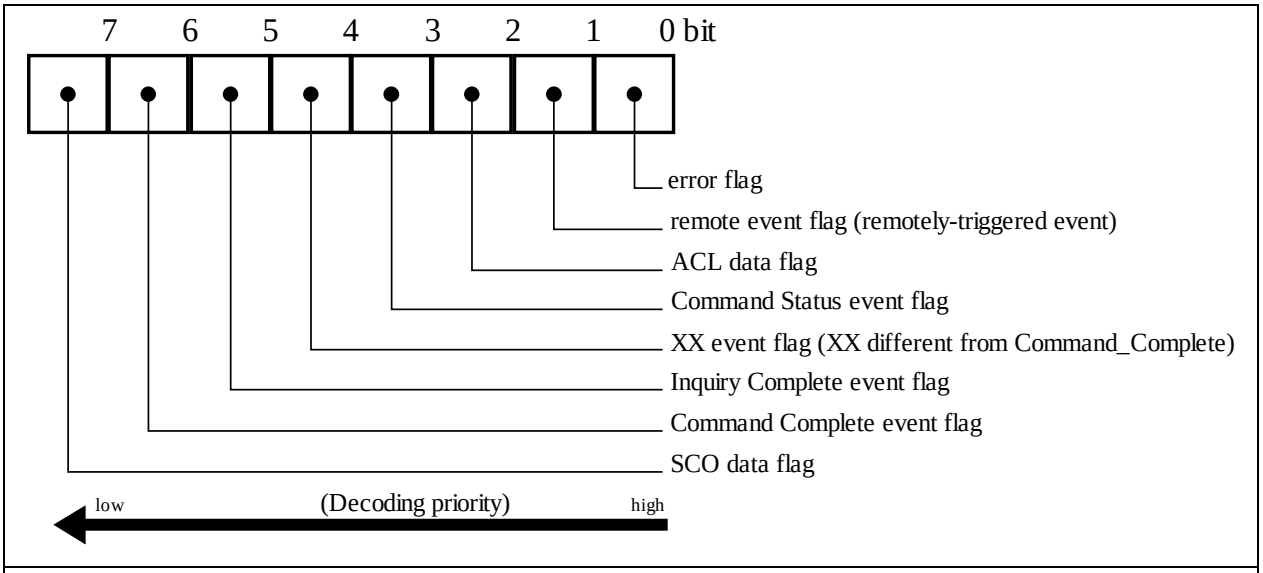


Fig. 5.15: "dcd_flags" contents

The remote-event flag is the next to be checked. Initially it is clear; when no specific pattern is expected and a remote event begins to arrive, the decoding subroutine sets the corresponding flag so as to schedule the proper decoding of the remote event (via the binary tree). If the remote event does not match any of the implemented ones, the subroutine checks whether this is the tip of an ACL data packet, after all. If this is the case, the ACL data flag is set to enable further decoding (also via the binary tree). On the contrary, the four following flags represent the "expected" events and -thus- are set during the decoding preparation process, examined previously. This means that apart from filling the event masks in the SRAM, one or more of those four flags of the dcd_prep register are also set during decoding preparation. The "XX" in the XX event flag represents the whole set of possible events *minus* the Command_Status, the Command_Complete and the Inquiry_Complete event (i.e. the command-specific events like Remote_Name_Request_Complete event etc.). The Inquiry_Complete event -even though belonging to those command-specific events- is handled separately due to its peculiar nature (i.e. multiple instances of this event can be received with only one Inquiry command, one event per discovered device). Also, for inner representation purposes and because of their especial structure,

Command_Status and Command_Complete events reserve their own flags. Finally, the SCO data flag is similar to the ACL one but is concerned with the reception of SCO packets[50].

Having each of these flags set, causes the decoding module to serially use the respective masks for byte-to-byte comparisons. For this reason, all the flags are checked in strict order from right to left (least significant bit first). If an error has occurred decoding is disabled (since the error flag is checked first). By placing the remote-event flag in higher priority than the rest events, two goals are achieved:

1. after issuing a command to its HC and while waiting for specific events in reply (i.e. some of the flags: Command Status, Command Complete, Inquiry Complete, XX event will be set), the Host may as well receive events from a remote Host (without prior notice). Making checks for remote events first, will not confuse the local Host which is expecting some other event(s) instead and -also- valuable time is gained by avoiding to check all the rest cases first.
2. HCI transport (here: the UART) is often apt to environmental disturbances; cable movement, electrostatic phenomena etc. make the UART "read" false bytes which are actually **noise** on the transmission lines. By giving such a high priority to the remote event flag, we manage to make the system **tolerant** to such kind of noise since the remote-event code segment of the subroutine makes provisions for noise compensation. This feature makes the rest code segments of the decoding process (the Command_Complete event, for instance) to work undisturbed.

5.3.2.3 Error handling in decoding

When an erroneous byte is received during the decoding of events from the Bluetooth module, the error flag becomes set and stays that way until the whole event has been received (and -probably- discarded). Yet, there is no way to exactly know when all the bytes from the erroneous packet have been received for the error flag to be cleared. This is so because, generally speaking, when the response is erroneous, there is no guarantee that the number of received bytes will be the one expected (according to the Bluetooth Specification v1.1). This element makes error handling in the BlueApplE a "best-effort" task since it does not guarantee completely error-proof decoding of all Bluetooth events[51].

[50] Even though provisions have been made for supporting SCO traffic, no handling of SCO packets has been implemented due to constraints in development time and due to the fact that the Bluetooth Module used does not inherently support SCO traffic (i.e. in terms of HW).

[51] For instance, if an error is detected in the middle of an Inquiry_Complete event reception, then following Inquiry_Complete events or even an Inquiry_Result event (signaling the termination of the inquiry period) may be ignored even though they are correct!

The concept behind error handling is as follows: in a sequence of incoming events, some packets present some inconsistency in their byte fields. The purpose of error handling is to isolate and discard only those erroneous packets while continuing to receive the rest of them normally. Towards that end, the 16-bit Timer/Counter1 of the AVR has been used. The maximum size for an event is 256 bytes (Remote_Name_Request_Complete). During event reception, a system register counts the number of already received bytes. Should an error occur, this number is subtracted from 256 and a **worst-case estimation** is formed on the number of bytes remaining for the current event. This estimated value (effectively converted to its corresponding *byte time*) is loaded[52] in the OC1A register, an output compare match register[53] of the Timer/Counter1. Then, the TC1 (which is initially cleared) starts to count up until it reaches the value stored in OC1A. During this period of time, the error flag is continuously set and -ergo- all incoming bytes are discarded, in a process called *decoding stalling*. When the TC1 reaches the value stored in OC1A, an internal interrupt is generated. The corresponding ISR clears the error flag, resets and freezes the TC1 and refreshes all decoding registers so that decoding can proceed normally. A slight variation to this general pattern exists when the event presenting an error is a Command_Status one. These events are statically 7 bytes long, so instead of subtracting the number of received bytes from 256, we do so from 7. Of course, in this case there is no estimation but rather an exact computation of the stall time needed[54].

Note that under the term "error" are included all byte errors regarding header fields of a packet. Errors in the parameters of an incoming event are not taken into account. However, integrity and validity checks on those parts are run separately and in a per-specific-event basis. Note also that this scheme is not applied in data packets since their size can be as large as 64 Kbytes (though only 672 bytes are allowed by the specific Bluetooth Module).

5.3.3 Connection management

[52] In fact, the value finally loaded is the equivalent 'byte time' of the remaining bytes; by taking into account the time needed to actually load and enable the TC1, the time for entering and exiting the ISR on compare match and various prescaler issues (TC1 prescaler is used), the initial 'byte time' is practically reduced.
[53] For more details on TC1 and the output compare match register see [17]: Timers/Counters
[54] For the time being, the described error handling routine counts properly only at 57.6 kbps, which is the default baud rate. Yet, it can be easily upgraded to work for all baud rates, e.g. through the use of a LUT including entries with delays for various baud rates.

The internal representation and management of HCI commands and events has been explained in the previous subsections. Yet, discussion here will be focused on the special provisions made and the actions taken regarding the subset of those implemented commands and events responsible for connection attachment and detachment, since they are the cornerstone of Bluetooth functionality. The commands are: *Create_Connection*, *Accept_Connection_Request* and *Disconnect* while the events included are: *Connection_Request*, *Connection_Complete* and *Disconnection_Complete*. A typical packet exchange scheme is the following one:

1. the local Host sends a Create_Connection command to the remote Host (after it has performed one or more Inquiry commands),
2. the remote Host receives a Connection_Request event,
3. if it wishes to connect to the local Host, it responds with a Accept_Connection_Request command; otherwise, it sends a Reject_Connection_Request command or simply lets the connection request time out,
4. the local Host receives a Connection_Complete event with its Status field signaling a successful or an unsuccessful connection establishment (in which case the reason is also given, e.g. time-out),
5. either the local or the remote Host reserves the right to terminate the connection at any time it sees fit by issuing the Disconnection_Complete event.

As mentioned in chapter 3, each Bluetooth Host can simultaneously support up to 8 ACL connections. The BlueApplE system has been built to actually support 8 ACL connections even though the Bluetooth Module used supports only one. For every active connection a minimum of two pieces of information must be saved inside the Host: the Connection Handle (2 bytes) of the connection and the BD_ADDR (6 bytes) of the remote device, for a total of 8 bytes per connection. These parameters are stored inside the SRAM, resulting in reserving 64 bytes in it. Every time the Host needs to send a packet over a specific connection, it copies the Connection Handle from the corresponding SRAM entry to the packet. In order to find the starting address of this Connection Handle, the structure utilized is a flag-offset register ("ACL_conn_off_flags"). An instance of this register is shown in figure 5.16.

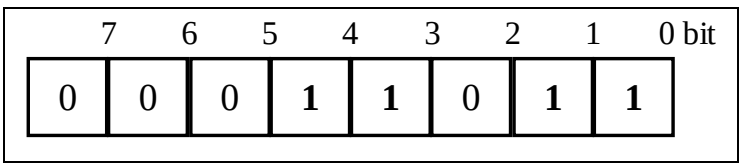


Fig. 5.16: Instance of "ACL_conn_off_flags" contents

The purpose of this register is dual: firstly, it provides a means of swiftly checking whether any connections are active or not (simply by comparing its value with '0'). Secondly, each bit of this register acts as an offset for the above-mentioned connection fields in SRAM. If for instance, five connections have been set up and the third one has been terminated after a while, the third set of 8 bytes of the connection-allocated space in SRAM will be invalid whereas the first, second, fourth and fifth set of 8 bytes will hold valid Connection Handle and BD_ADDR fields. In this case, the register will have its bit 2 cleared and its bits 0, 1, 3 and 4 set. Then, if the Host tries to send data over, say, its third active connection, it will ignore the third 8-byte SRAM entry (which is now invalid) and will correctly use the fourth 8-byte entry to send its data. To do so, the Host scans the register from right to left (least significant bit first); for every bit equal to '0' it shifts a pointer 8 bytes in SRAM and for every bit equal to '1', it decreases by one unit the number of times it needs to shift. This "number of times" is actually defined by an **active-connection index** selected by the external user via the Input Interface functionality, discussed in the next subsection. When this number reaches '0', the correct bytes in SRAM have been found and the Host issues the data packet which is filled with the Connection Handle of the specific SRAM entry. The relation between the register and the SRAM connection entries is graphically depicted in figure 5.17.

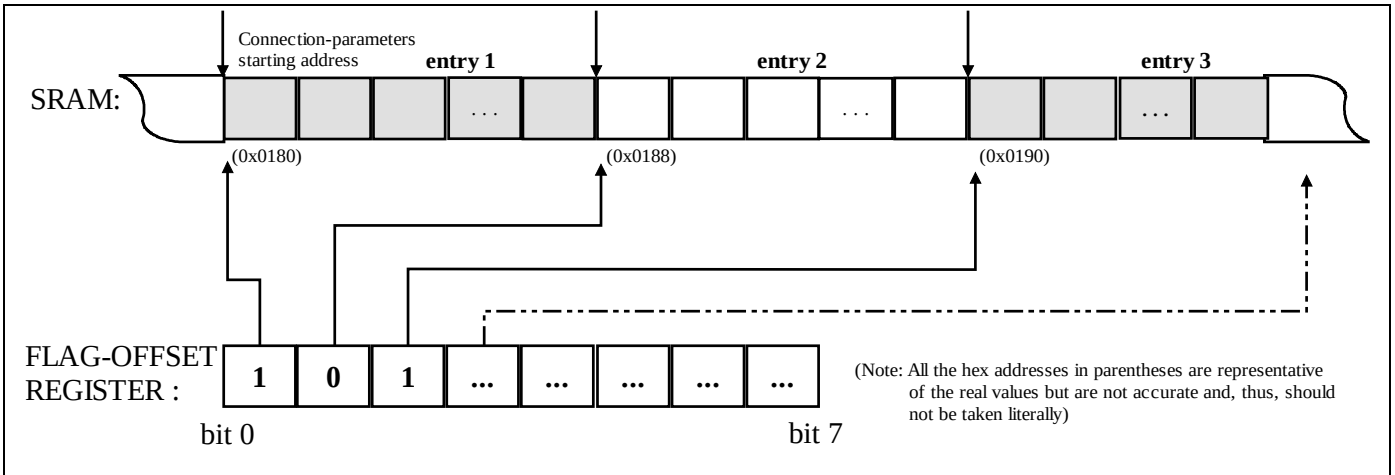


Fig. 5.17: Instance of running connections and respective system structure

The flag-offset register along with the appropriate SRAM slots constitute an extremely effective way of managing ACL connections. This can be demonstrated in the following scenario: a new connection is to

be established in the system just described. The Host scans the flag-offset register from right to left until it finds a bit equal to '0'. For every non-zero bit, a pointer is shifted 8 bytes in SRAM. When a zero bit has been found (here: the third one), a free entry in SRAM has automatically been found (here: SRAM connections starting address + 2 * 8 bytes) and the new-connection parameters are written in it (while the corresponding bit is set). In case a Disconnection_Complete event has been received, the procedure followed is exactly the opposite: the above event contains the Connection Handle of the connection to be terminated. By sequentially comparing this Connection Handle with the ones in the SRAM connections entries, a counter is increased (which was initially cleared) until a match occurs (say, counter = '4'). Then, the flag-offset register is scanned from right to left until the 4th bit that is equal to '1' is found (bits with zero values are ignored). By clearing this bit, the corresponding space in SRAM is automatically freed and can be used for future connections. Obviously, on the part of the Host issuing the Disconnect command, the active-connection index matches the value of the above counter so the corresponding bit in the flag-offset register can be directly cleared (i.e. no Connection Handle comparisons need to be performed).

In a nutshell, when a Accept_Connection_Request command is issued from a remote device, both the local and the remote Host act according to the algorithm depicted in figure 5.18. When a Disconnect command is issued by either of them, the procedure followed is the one depicted in figure 5.19.

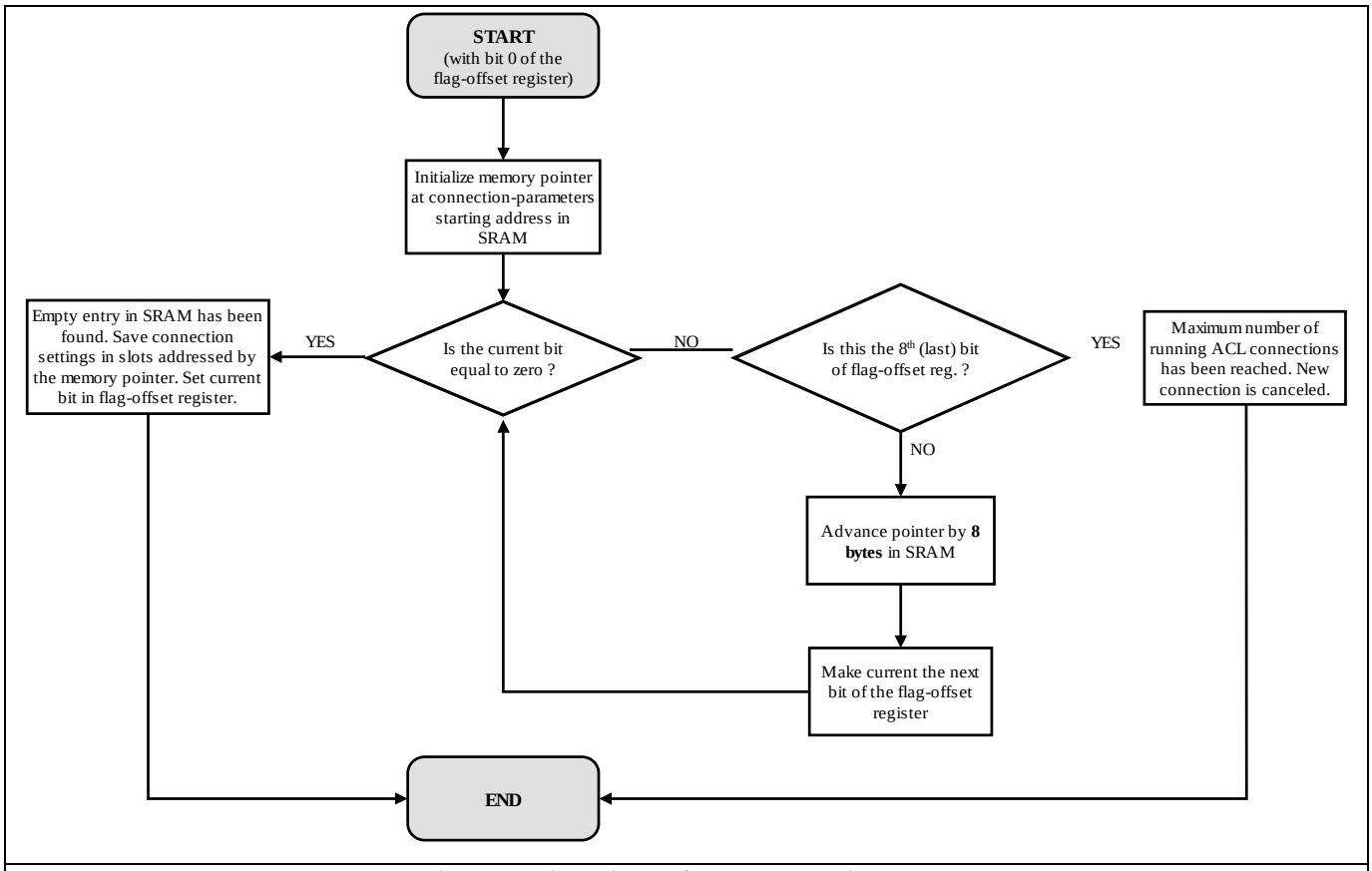


Fig. 5.18: Flow chart of ACL connection setup

It is noted here that the Host cannot simultaneously process multiple connection requests even though it can support multiple connections. Also, in all above operations, internal organization is such that even though different procedures take place in the local and remote Host during connection setup, handling of both local and remote systems is uniform and seamless.

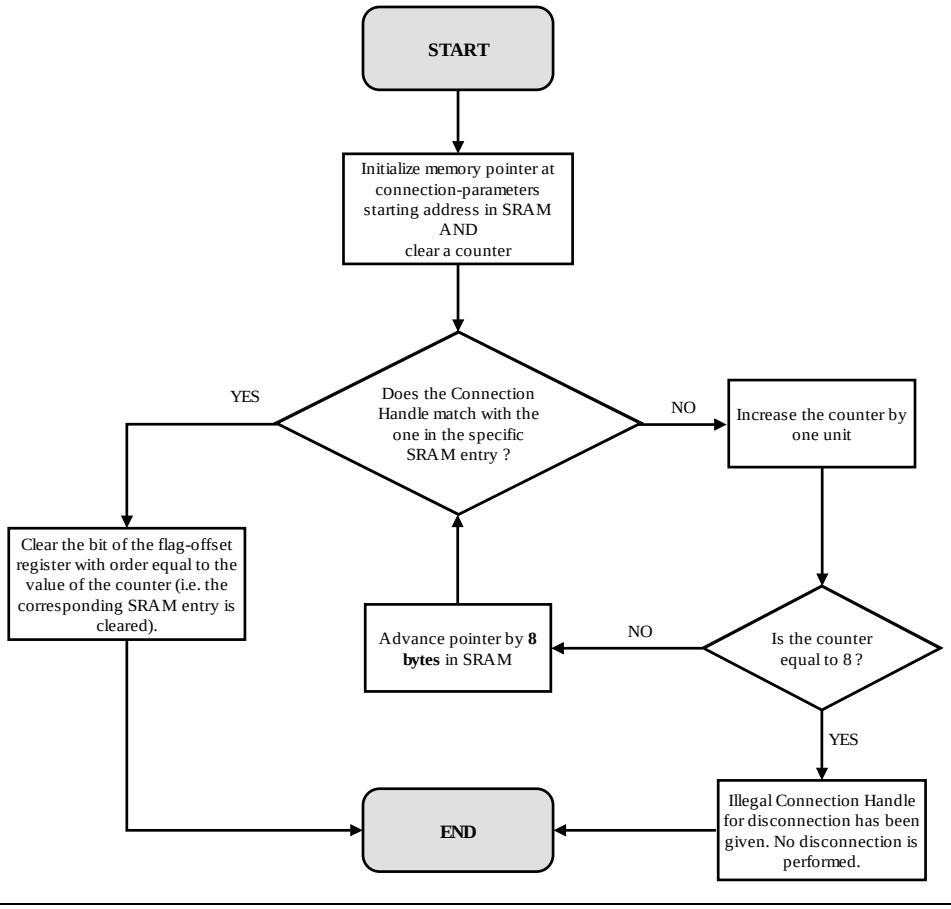


Fig. 5.19: Flow chart of ACL connection termination

5.3.4 Input Interface: Software-enabled features

In a previous subsection, the HW structure of the Input Interface has been deployed. But merely providing a set of multiplexed, debounced buttons as inputs is not enough. The functionality given to those buttons has made the unleashing of all implemented Bluetooth (and system) features possible. In figure 5.20 a description of those functions of each button is given. Buttons '1', '2' and '3' are irreplaceable in the BlueApplE. They are built according to the "fast-forward" / "rewind" notion used for navigation in many device menus. Button '3' defines the mode of operation i.e. depending on the value selected by this button, different actions are accomplished:

Button number	Function
I	Command mode: "0": select next command "1": select next inquired device "2": select next active connection

	"3": select next baud rate for external UART (see chapter 6)
II	Command mode: "0": select previous command "1": select previous inquired device "2": select previous active connection "3": select previous baud rate for external UART (see chapter 6)
III	Change command mode (cycle values: "0" through "3")
IV	Issue selected command / Change the baud rate of the external UART (see chapter 6)
V	[RESERVED FOR FUTURE USE]
VI	[RESERVED FOR FUTURE USE]
VII	Clear port A and port C error messages (i.e. set to normal display)
VIII	Enable / Disable the BlueBridge test application (see chapter 6)

Fig. 5.20: Functionality brake down of Input Interface buttons

- With the *command mode* set to **'0'**, button I selects the next HCI command with respect to the table seen in fig. 5.10; actually, the *command index* is increased by one unit. Button II selects the previous HCI command from the same list (likewise, the *command index* is decreased by one unit).
- When performing an Inquiry command, information from multiple devices (in the vicinity) are accumulated and stored inside the AVR. With the *command mode* set to **'1'**, buttons I and II scroll (up and down, respectively) through all the inquired devices and select one of them as active. When we need to establish a connection, it will be established between the local device and the device selected from the above inquired-device list.
- With the *command mode* set to **'2'** and having multiple connections running, buttons I and II select a specific one among them for use. All HCI commands referring to a specific connection (e.g. Disconnect command) will be applied to the one connection specified by the buttons I and II (also up and down scrolling).
- On pressing button IV, the command selected through buttons I and II (with *command mode* set to **'0'**) is loaded up and transmitted over the HCI if *command mode* is not equal to **'3'**. In that case, the command sent is a *write configuration* command sent to the external UART for changing its running baud rate (see chapter 6 for more details).
- Obviously, button III cycles through the various command modes from '0' to '3' enabling different functions for buttons I and II.

Operations can be easily assigned to the remaining, unreserved buttons, while functionality of button VIII will be further elaborated in the next chapter. For conforming to the table of fig. 5.3, it should be

noted that pins PD5 and PD4 of the AVR are connected to LEDs for displaying the active command code on-board the BlueApplE. Lastly, a key feature of the button functionality which has been discussed before is their reconfigurability. The SW driving those 8 buttons is structured in such a manner that it can be altered very easily. Also, this SW module alters no system registers; it only reads them, instead. So, changing part or whole of the Input Interface SW will not affect the BlueApplE functionality (except for the user handles, that is).

5.3.4 Error & Informational messages

The underlying HW structure for exporting various system messages such as errors, parameter values, operational details, active states etc. is the Output Display, a set of two groups of LEDs. Each group includes 8 LEDs being possible to display exactly one byte at any time. As previously discussed, port A is labeled **"Index & Error Display"** while port C **"Error Details Display"**. The reason is that port A concisely displays all the system active settings, such as the currently selected command or inquired device or active connection (as seen in the previous subsection). It also displays the event type when an erroneous event under reception is detected Port C, on the other hand, is more like an auxiliary display, outputting information in those cases that port A is insufficient. For instance, when port A displays the event type of an erroneous byte, port C displays the number of the first byte in which the error occurred so as to give a more detailed view of the error. In figure 5.21 an analytical table of values for port A and port C are given. It should be noted that in most cases and functions described through this whole chapter (and in the next one), both ports provide various data, in concurrence.

Port A	Port C	Description
Input Selection		
0x01 – 0x1B	-	Command Selection range [1 - 27]; 27 commands are implemented so far (access with btt3 = 0, btt1/btt2 = next/previous).
0x00 – 0x0A	-	Inquired-Device Selection range [0 - 10]; data from 0 to 10 inquired devices can be stored in SRAM (access with btt3 = 1, btt1/btt2 = next/previous).
0x00 – 0x08	-	Active-Connection Selection range [0 -8]; a maximum of 8 ACL connections is supported (access with btt3 = 2, btt1/btt2 = next/previous).
0x00 – 0x0F	-	xUART B.R. Selection range [0 - 15]; the MAX3110E datasheet defines 16 different baud rates (access with btt3 = 3, btt1/btt2 = next/previous).
-	0x0Y	On xUART baud rate change success, display in port C the new baud rate (access with btt4).
0xYY	0xYY	Set port A and port C displays to their values prior to any error message (access with btt7).

0xBE	0x01	BlueBridge error; local side attempted BlueBridge initialization with no ACL connections running (btt8)
Command Issue (through button '4')		
-	0xC0	Device issued Create_Connection command while no inquired devices exist (no data available).
-	0xC1	Device issued Disconnection command while no ACL connections are running.
-	0xC2	Device issued Accept_Connection_Request command without receiving a Connection_Request event.
-	0xC3	Device issued Remote_Name_Request command while no inquired devices exist (no data available).
Event / Data Decoding		
0x04	0xYY	Remote event error. Port A displays the position of the erroneous byte in the event (e.g. 0x05).
0x40	0xYY	Disconnection_Complete event failed on remote side due to reason (i.e. status field) displayed in port A
0x41	-	Disconnection_Complete event failed on remote side due to error in given Connection Handle.
0xBE	0x02	BlueBridge error; multiple BlueBridge initialization attempts made by remote side.
0xBE	0x03	BlueBridge error; invalid BlueBridge parameter given by remote side (for "control" packet).
0xDF	0xYY	BlueBridge warning; "raw" data packet received not part of the active BlueBridge. Port C displays the "maverick" byte; this byte is not propagated over to the xUART.
0x02	0xYY	ACL-data-packet / BlueBridge error. Port A displays the position of the erroneous byte in the event; if error exists in byte 8 or 9, a byte with an invalid CID was trying to pass over the BlueBridge.
0x0F	0xYY	Command_Status event error. Port A displays the position of the erroneous byte in the event.
0x03	0xYY	XX event error. Port A displays the position of the erroneous byte in the event.
0x19	-	Inquiry_Result event error. When more than 10 (remote) devices respond to an Inquiry command, the SRAM-space assigned for inquiry-acquired data becomes full and, in order to avoid a memory overflow, new device data (after the 10th responding device) overwrite the first entries (wrap-around). The number of inquired devices is reset to '1'.
0x19	0xYY	Inquiry_Result event error. Port A displays the position of the erroneous byte in the event. (Actually, only the static field "Num_HCI_Responses" is checked).
0x20	0xYY	Connection_Complete event failed on remote side due to reason (i.e. status field) displayed in port A.
0x21	-	Connection_Complete event failed on remote side since 8 ACL links are already running
0x0E	0xYY	Command_Complete event error. Port A displays the position of the erroneous byte in the event.

Legend:

- "btt" stands for button
- "-" in a port A or port C entry means that the corresponding port is not affected by the specific error or message

Fig. 5.21: Overview of Output Display messages

A few clarifications must be made on the above table:

- Command Selection is directly related to the number of implemented HCI commands and can be increased with new additions. Inquired-Device Selection has been statically limited to 10 devices (through the constant MAX_INQ_NO). There is no theoretical maximum number of devices that can be inquired

by a device; however, 14 bytes are needed in SRAM for storing the data acquired by a single inquired device. This puts a practical limit to the devices "remembered" since the SRAM cannot possible include all of them. Yet, if more memory is available, capacity for new devices can be increased by simply increasing the number of MAX_INQ_NO; *no further changes* in the design are required. Also, Active-Connection Selection is limited to 8 simultaneous ACL connections. In this case, however, the limit is set by the BT spec. itself which defines that a maximum of 8 ACL links can be up and running on any Bluetooth device at any time.

- For the meaning of the terms "raw data" and "control" packet as well as the various BlueBridge errors and warnings, refer to the following chapter, since they are involved with the BlueBridge operation.
- When an error occurs and the *reason* is displayed in port A, this reason is actually the 1-byte "status" field carried by most of the HCI events describing the event status. A value of '0x00' exhibits a normal (error-free) execution while non-zero values represent different error codes. For a detailed description of the various status values, see [4]: Part H1, List of Error Codes.

6

BlueBridge Application

6.1 BlueBridge overview

In the previous chapter, a comprehensive overview of the BlueApplE system has been given. The system HW and SW structure has been explained and its capabilities have been stressed out. However, the thing missing has been a specific application that would put all this effort to practice. Towards this end, we have also deployed the BlueBridge application. As discussed in chapter 1, BlueBridge has a dual role: firstly, it constitutes a test application for **display** and **debugging** purposes of the BlueApplE functionality and, second, it provides an **intriguing "instance"** of the Bluetooth technology; to fluently depict its potential and overall performance. With respect to *timing* and *financial* constraints, the application attempted is the implementation of a "wireless UART", so to speak.

Many high-level applications communicate with peripheral devices through the UART I/F, which has been -at least till the USB arrival- a proven and easy way of connection. The wide acceptance of serial communications has been even pinpointed by the Bluetooth SIG and is -thus- mirrored in its design choice to include the RFCOMM in the protocols of the Bluetooth stack. For these and other reasons, the BlueBridge application has been chosen to be implemented on-board the BlueApplE. It effectively supports an over-the-air connection between two higher-level hosts (applications, devices etc.) that used to communicate through a *wired* UART I/F. The BlueBridge makes **no assumptions** on the host nature, the kind of data exchanged or the stream behavior (e.g. bursty, continuous, sporadic etc.) whatsoever; the only restriction is that mere UART is used since full RS-232 is not supported. It must be stressed out -however- that this implementation has nothing to do with the RFCOMM protocol. No part of this protocol has been built for enabling serial communication so no RFCOMM-specific features are existent such as multiplexing of various streams over the same channel etc., as seen in chapter 3. The reason for that is again limitations in time and in the material used (e.g. there is no way that the RFCOMM protocol can fit inside the AVR used). Since the BlueBridge has been pursued to be an application located in an embedded system, special provisions have been made and implementation has been clearly a custom one.

When this thesis was initiated Bluetooth applications were rare and "hesitant"; a wireless UART would have definitely been a "killer-app". Given this Bluetooth explosion that took place in 2002, though, the importance of such an application diminished as various Bluetooth "fabrics" found their way to the market. Examples of such PC-mounted devices, ASICs or ASSPs either implementing or embedding wireless serial communications have been presented in chapter 2; they are only the tip of the iceberg. In

any case, BlueBridge has been designed to be a pilot application rather that a so-called killer-app. To get an idea what this BlueBridge is all about, a block diagram is depicted in figure 6.1.

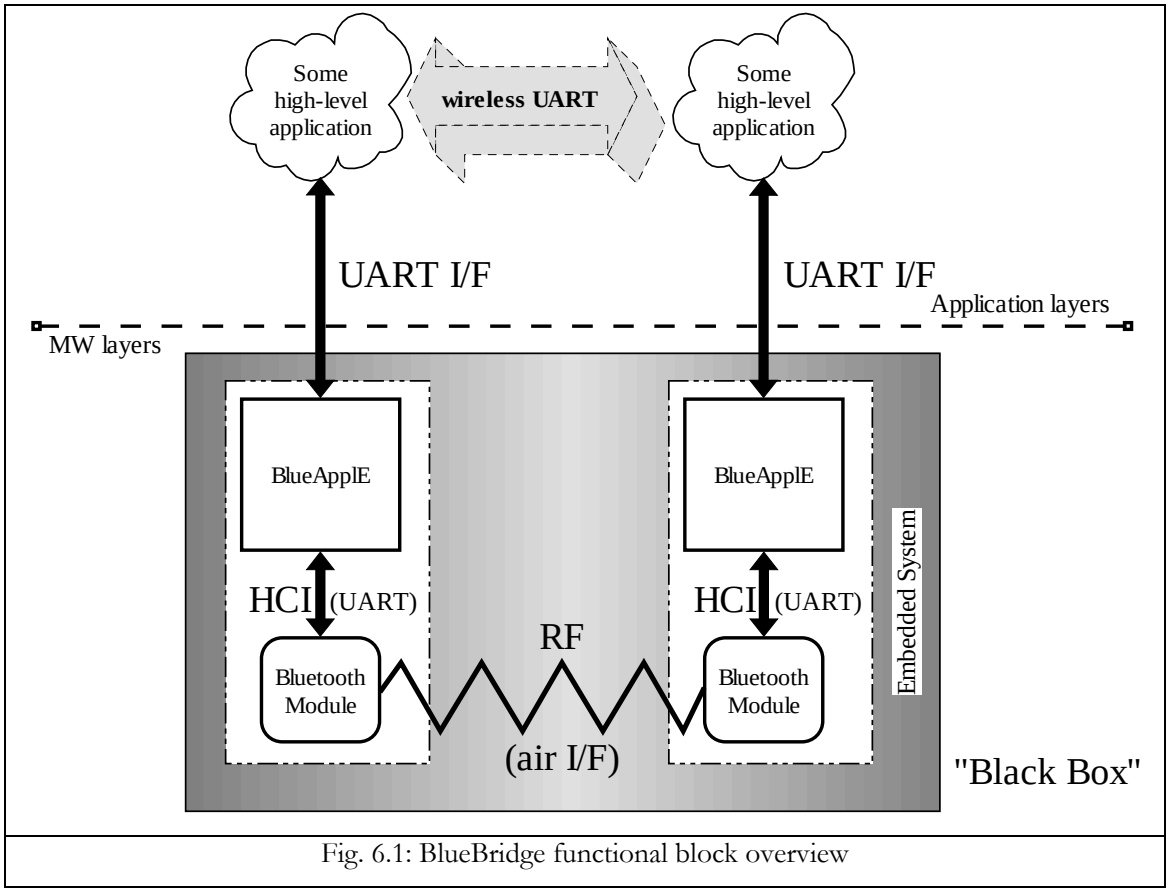


Fig. 6.1: BlueBridge functional block overview

A detailed description of the design and implementation steps of BlueBridge is included in the following sections.

6.2 Hardware specifics

Each of two BlueApplE boards is connected to a Bluetooth Module as described in chapter 5. In order to provide a UART I/F to a high-level application, each BlueApplE needs a UART module. The problem is that the AVR used, includes only one UART module, the one used as the physical transport for the HCI. For this reason, an external UART (hereon, **xUART**) has been utilized in the implementation. This is the **MAX3110E** chip from Maxim-IC and even though not many alternatives existed, it has proven to be an exceptional chip [24]. Its pin-out is depicted in figure 6.2 and its features are:

- Integrated RS-232 Transceiver and UART in a Single 28-Pin Package
- SPI/QSPI/MICROWIRE-Compatible μC Interface
- Internal Charge-Pump Capacitors (No External Components Required)
- ESD Protection for RS-232 I/O Pins:
  - ±15kV - Human Body Model
  - ±8kV - IEC 1000-4-2, Contact Discharge
  - ±15kV - IEC 1000-4-2, Air-Gap Discharge
- Single-Supply Operation: +5V
- Low Power
- 600μA Supply Current
- 10μA Shutdown Supply Current with Receiver Interrupt Active
- Guaranteed 230kbps Data Rate

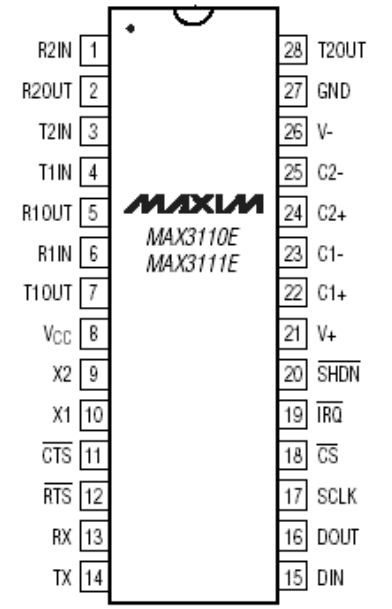


Fig. 6.2: MAX3110E external UART (Narrow DIP package)

It also includes an 8-byte deep FIFO, supports 9-bit words and a full set of RS-232-relative signals (CTS, RTS) and flags (framing error etc.). Of all the above features, the most crucial ones have been its extremely **low current** requirement, which is optimal for the low-power operation of the Bluetooth system of ours; also, its **high data rate** (up to 230.4 kbps)**,** its **TTL-level voltage** of 5V (allowing for uniform voltage network across the circuit) and the fact that **RS-232 line drivers are internally built** through the use of charge-pump capacitors (i.e. no external circuitry is required). The last two features make it very space- and cost efficient; a perfect solution -indeed- to incorporate in an embedded application as the one at hand.

I/O in the chip is channeled through the SPI (mode 0: CPOL = 0, CPHA = 0)[55] I/F standard. The AVR reserves specific I/O pins for SPI communication: PB4 (/SS, Slave Select), PB5 (MOSI, Master Output Slave Input), PB6 (MISO, Master Input Slave Output) and PB7 (SCK, SPI Clock). That is why these pins have been left untouched by other operations, as seen in the table of figure 5.3. Just like in the case of the built-in UART module, the AVR fully supports the SPI module, allowing for seamless operation with minimum configuration[56]. A block diagram of the interconnection between the AVR and the MAX3110E is depicted in figure 6.3.

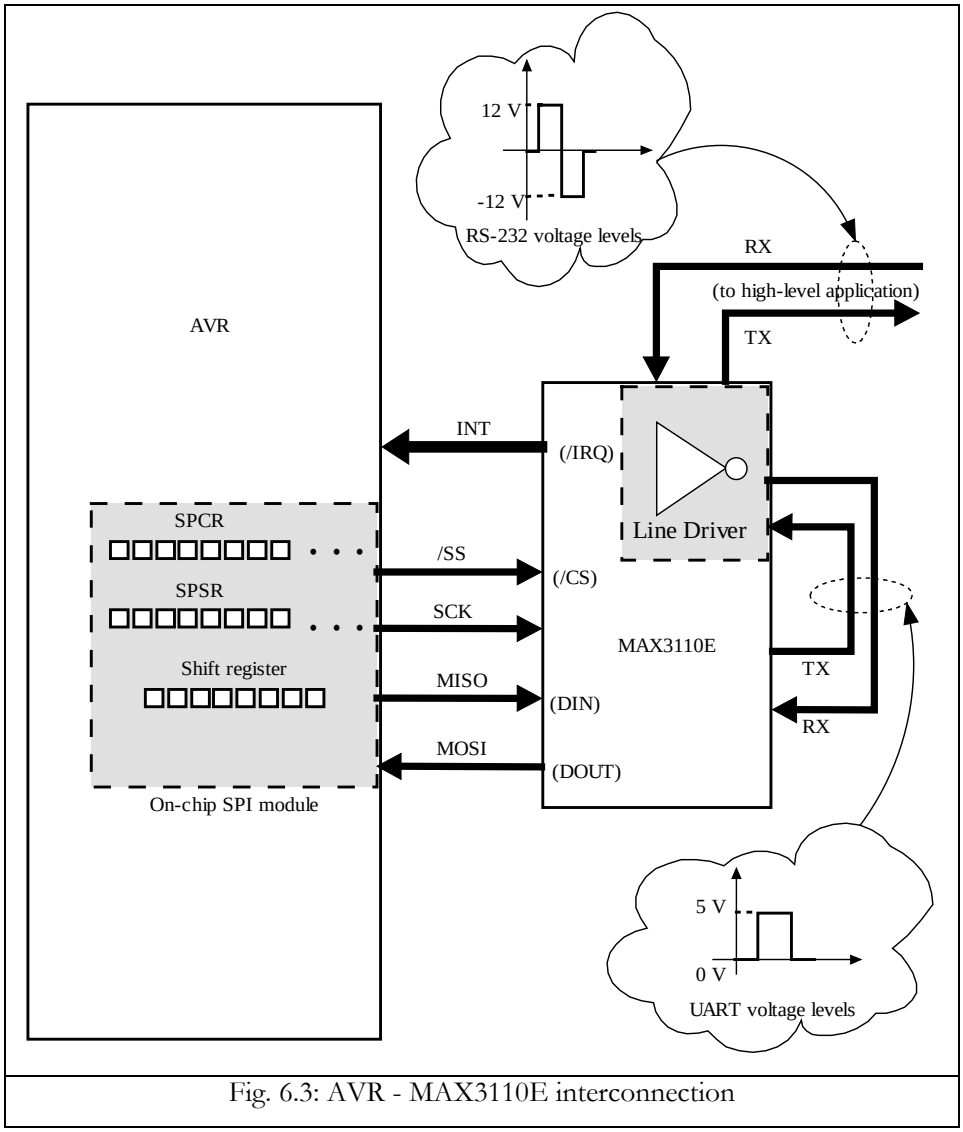


Fig. 6.3: AVR - MAX3110E interconnection

[55] The SPI standard provides for 4 modes of operation, the four combinations of the 1-bit quantities CPOL and CPHA, both referring to the SPI Clock (SCK). CPOL stands for Clock Polarity and CPHA for Clock Phase.
[56] For more details on the functionality of the SPI provided by the AVR, see [17]:SPI.

On each BlueApplE, the MAX3110E chip is responsible for sending data from an outside application to the AVR (for over-the-air transmission) and for forwarding data from the AVR towards the application. The chip can be configured to generate an *interrupt* each time a new byte is being received (through the "/IRQ" pin). If this pin is connected to a corresponding INT pin of the AVR, an ISR is executed every time there is incoming traffic on the xUART. Referring once more to the table of figure 5.3, it can be seen that PD3 (i.e. INT1) of the AVR has been bound to this task. The actions taken every time the ISR is executed will be discussed in the next section. The gain from this interrupt feature of the xUART is obvious: CPU latency is kept to a minimum by avoiding periodic polling on the chip buffers.

6.3 Software specifics

Having seen how the MAX3110E chip is physically connected to the BlueApplE system and the functionality it provides, it is now time to see how it is tuned up with the rest of the system in order to enable this wireless UART.

6.3.1 External UART configuration setup

Every time the BlueApplE system is powered or reset, both the AVR and the xUART need to be configured. As seen in the previous chapter, the AVR initially writes its internal SRAM and assumes some operational states; it is also the AVR's task to properly set up the xUART. The xUART is controlled by two pairs of 16-bit long commands: *write/read configuration* and *write/read data*. The first pair of commands is used for setting the various UART parameters of operation such as the baud rate, the data word length (8 or 9 bits), the parity enable, and the 8-word receive FIFO enable. This pair is also used for setting other internal parameters such as the INT enable (seen above). More self-evidently, the second pair of commands is meant for send and receiving data over the xUART. Since all four commands are 16-bit long and the AVR supports 8-bit transmission through the built-in SPI, two (8-bit) SPI transmissions are needed for each command. The xUART follows the SPI convention of providing a bidirectional data path for writes and reads. Whenever data is written, data is also read back for speeding operation over the SPI bus and -thus- speeding the xUART. For a detailed description of the 16-bit SPI read and write methods applied to the xUART, see Appendix D. Suffice to say that all 16 bits of a *write/read configuration command* contain settings of the xUART operation. Similarly, from the 16 bits of a *write/read data command*, the low 8 bits constitute the byte respectively sent to or received from the

xUART, whereas the high 8 bits contain status flags (transmit buffer full, data in FIFO available) and traffic flags (framing error, parity bit, CTS, RTS).

For the purpose of this thesis, a specific write configuration command is sent to the xUART, concisely holding the following features:

- xUART receive FIFO is enabled
- FIFO interrupt enabled (as seen above)
- 8-bit words are used (1 stop bit, no parity bit)
- baud rate: 57.6 kbps (default)

According to the datasheet, the MAX3110E chip presents oscillations for some time after power up or reset, so the algorithm of figure 6.4 is required for properly setting up the xUART. On success, the AVR exits the loop and notifies for proper xUART configuration by turning the LED connected to PD6 on (refer to the table of figure 5.3).

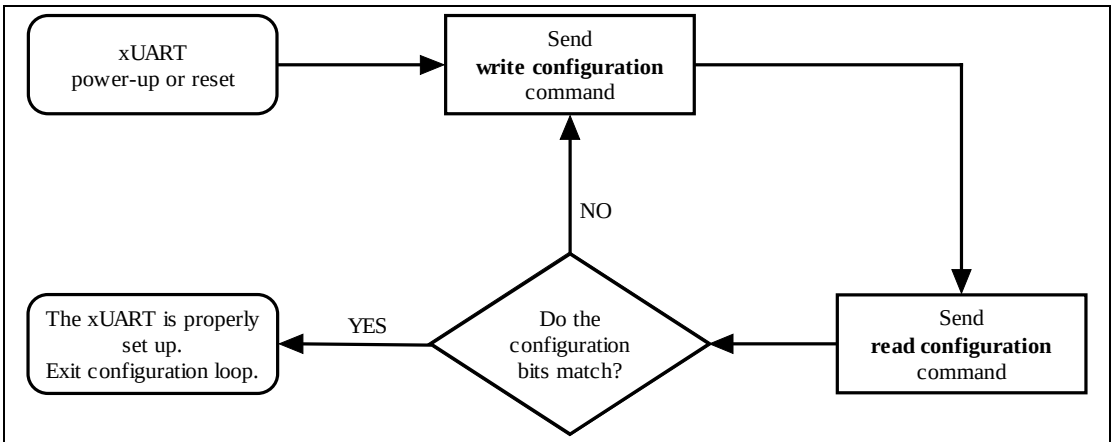


Fig. 6.4: xUART configuration setup

In the previous chapter, when describing the functionality of the Input Interface buttons, the setting of *command mode* to '3' had not been covered (table of figure 5.20). When this command mode is selected, we can change the baud rate of the xUART through the buttons '1' and '2' (up and down, respectively). Supported baud rates are all standard rates from **300 bps** to **230.4 kbps**. By pressing button '1' or button '2', higher or lower baud rates are displayed in port A. Then, by pressing button '4', the currently selected baud rate is made active and port C confirms the successful change by displaying this new rate.

In this way, the xUART is easily configured to support all wanted baud rates. What is actually done inside the AVR, is the execution of the above algorithm once more but with a sole parameter change in the transmitted configuration command (the one parameter responsible for the value of the baud rate).

6.3.2 BlueBridge core design

Having seen the xUART basic elements of operation, it is now time to concentrate in the BlueBridge SW core. As noted in chapter 5, the AVR internal SRAM holds all the HCI command packets and various HCI event packet masks. However, this has not been accurate in the sense that SRAM also holds a plethora of other information such as the data of the inquired devices, the data of all currently active connections etc. and many more sparse byte-fields with system data needed for smooth BlueApplE operation. Apart from all these, in view of the BlueBridge application, additional space has been reserved for **ACL data packet masks** just like HCI event masks, although in this case their functionality is somewhat different. For the purposes of this thesis, ACL packets of static lengths have been used (with payload equal to <u>one</u> byte only). The ACL mask stored in SRAM is actually an ACL packet "empty shell". In this sense, length fields and ACL flags are statically written but dynamically changing fields such as the actual payload and the Connection Handle are to be masked with different values every time. The structure of an ACL packet (shown in figure 3.17) is repeated in figure 6.5 for convenience and so is the structure of a L2CAP packet (shown in figure 3.13) in figure 6.6.

<table>
<tr><td colspan="8">0 4 8 12 16 20 24 28 31</td></tr>
<tr><td colspan="4">Connection Handle</td><td>PB Flag</td><td>BC Flag</td><td colspan="2">Data Total Length</td></tr>
<tr><td colspan="8">Data</td></tr>
<tr><td colspan="8">Fig. 6.5: HCI ACL data packet format</td></tr>
</table>

<table>
<tr><td colspan="2">Header</td><td>Payload</td></tr>
<tr><td>Length
- 2 bytes -</td><td>Destination_CID
- 2 bytes -</td><td>Payload
- (Length) bytes -</td></tr>
<tr><td colspan="3">Fig. 6.6: L2CAP_PDU format (CO)</td></tr>
</table>

As seen in figure 6.7, we choose PB_flag = "0b10" (i.e. beginning of L2CAP_PDU) and BC_flag = "0b00" (i.e. point-to-point broadcast to active slaves). Also, as discussed in chapter 3, the HCI packet payload must be a whole or a fraction of a L2CAP packet (L2CAP_PDU). By selecting a L2CAP payload size of 1 (actual) byte, the "Data Packet Length" field of the HCI ACL packet will be 5 bytes long: 2 bytes for the L2CAP payload size, 2 bytes for the CID and 1 byte for the L2CAP payload. The only fields remaining which are dynamically defined are the Connection Handle, which depends on which active ACL connection issues the data packet, and the low byte of the CID field which (as seen in chapter 3) is allowed to takes values in the range: 0x0040 – 0xFFFF. Whereas the CID field is meaningless to the HCI layer, different values of the CID are used for custom system purposes, explained below.

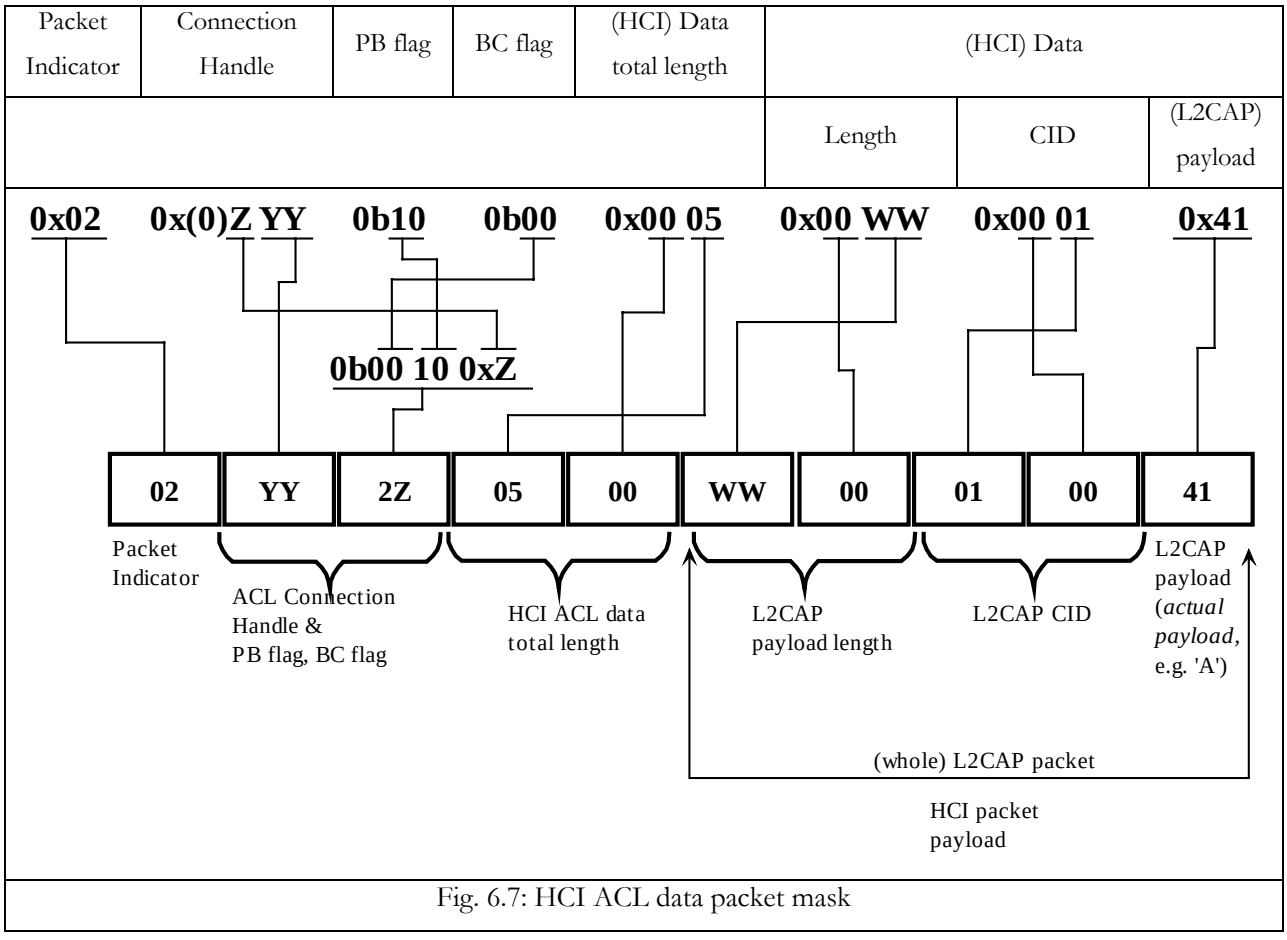


Fig. 6.7: HCI ACL data packet mask

Referring to the table of button functions (fig. 5.20) once more, we can see that button '8' activates a SW module, used for turning the BlueBridge application on and off. A prerequisite for starting the

BlueBridge is that an ACL link is already up and running. If not, an error and reason is displayed (see table of fig. 5.21 for details). If a connection is running, multiple button presses will result in consequent attachments and detachments of the BlueBridge. Apart from mere data (of the high-level applications) running over the BlueBridge, signaling information must also be transferred so that the BlueApplE system can control it. For this purpose two CIDs are used: '0x0040' and '0x0041'. The former marks an ACL data packet like the one in fig. 6.5 whose L2CAP 1-byte payload carries high-level application data (hereon "raw" data), while the latter carries in the 1-byte payload BlueBridge signaling data (hereon "control" data). The functionality is lively displayed in figure 6.8.

CID	Function
0x0040	*Control* data are carried in the L2CAP payload of the specific ACL data packet. If the L2CAP payload is: • 0x00: send the termination signal for the BlueBridge to the remote end. • 0x01: send the initialization signal for the BlueBridge to the remote end.
0x0041	*Raw* data are also carried in the L2CAP payload of the specific ACL data packet. The byte carried is an information packet flowing from one high-level application to another. NO PROCESSING of this byte is allowed.

Fig. 6.8: BlueBridge-utilized CIDs and functionality

With respect to the above table, when the BlueBridge application is to be initiated, the triggering BlueApplE system (say device A) sends a data packet like the one depicted in fig. 6.5 to the accepting system (say device B). The Connection Handle field ('0xZYY') will be filled with the Connection Handle of the active ACL link onto which the BlueBridge is being initiated (e.g. '0x001'). Also, the CID field ('0x00WW') will be equal to '0x0040' since the packet is meant for control, and the payload field will be set to '0x01' since the BlueBridge is to be initialized. In the same way, device A (or device B) can stop the BlueBridge (by a second press of button '8') by sending the same "control" packet but with the payload set to '0x00' for signaling the termination of the BlueBridge. It should be stressed out that, on reception of one of the two control packets, both A and B devices can manipulate the BlueBridge as equals, regardless of which one of them has initiated or terminated the application (i.e. no master – slave distinction exists). This provides a refined *symmetry* in operation which -in turn- simplifies BlueBridge handling by the external user. To achieve this symmetry, a kind of distributed system is shared between the two BlueApplE systems. Once more, referring to the table of the figure 5.3, PD7 has been reserved for switching a LED on, on BlueBridge attachment, and off, on BlueBridge detachment. The SW module that toggles the BlueBridge performs a final task before returning. It prepares the system for

raw-data transmission, i.e. it updates the CID field of the ACL data packet mask (in SRAM) with the value '0x0041' which stands for raw data. In so doing, any incoming bytes from the external application need only write the L2CAP payload field with the actual data and transmit the whole packet. The advantage is that precious time is saved (by avoiding memory traversing and writing) which is of the essence when the BlueBridge is running and the communication speed is high.

Till now, "control"-carrying data packets have been discussed. It is time to take a look at how the actual "raw data"-carrying packets are handled. When data from a high-level application are received in the FIFO of the xUART, the above-met /IRQ pin of the chip is pulled low and INT1 of the AVR activates an ISR. This ISR performs a *Read Data* command to the xUART and acquires the byte just received. It, then, writes the received byte in the position (in SRAM) of the L2CAP payload and transmits the whole ACL data packet over the air. On the side of device B, the decoding routine described in the previous chapter, receives and recognizes the ACL data packets as such. If they are "control" packets, they are handled internally in the way described above. If they are "raw data" packets, a different process is followed. On the received packet, checks are run for consistency in its fields. Special checks are run on the CID field which must strictly be equal to "0x0041" or an error message is generated (see table of fig. 5.21 for details) and data is not further forwarded. If no problem occurs, the received packet is stripped off of the actual 1-byte (L2CAP) payload which must now be forwarded to the xUART of device B. This is done by issuing a *Write Data* command (containing the received byte) to the xUART which -in turn- sends the byte to the remote high-level application.

Of course, the processes (in devices A and B) described above can be interchanged since the BlueBridge supports seamless **bidirectional** UART communication between applications. Also, when the BlueBridge is offline, **no forwarding** of "raw data" takes place; that is, even if ACL data packets are successfully exchanged over the air and up to the HCI level, they are "filtered out" by the BlueBridge-relative SW module. On the other hand, when the BlueBridge is online and either device A or B receives a BlueBridge initialization "control" packet, an error message and details are generated (see table of fig. 5.21 for details). A reason for sending twice an initialization packet may be packet loss over the air but this check has been included mainly for security reasons, in case a "monger" device in the vicinity (say device C) tries to intercept the exchanged application data or to take advantage of device A or device B resources. Last but not least, it should be noted that only two "control" packets have been introduced:

the initialization (0x01) and the termination one (0x00). The SW module structure is highly **modular** and can be easily updated to support more "control" packets. For instance, the number '0x02' could be reserved for baud rate change of the xUARTs. This may be the case when device A's xUART needs to increase its baud rate; in order to avoid a bottleneck on the other side (device B), it sends over a "control" packet with parameter '0x02' asking the remote xUART to also increase its baud rate.

7

Results & General Issues

7.1 BlueBridge results

By deploying this wireless-UART application, insight and in-depth understanding of the Bluetooth technology has been gained. The system has been seen as a "whole" and not as a set of dangling, meaningless HCI commands. The potential of the Bluetooth technology has been sufficiently pointed throughout the development of this thesis and the path for more Bluetooth-enabled applications has been paved. Even more importantly, the BlueApplE has proven to be a stable platform and also a very suitable one for the deployment of other Bluetooth applications, making it the potential cornerstone for many embedded, Bluetooth-enabled designs of the future.

It is worth mentioning that an attempt has been made for **remotely programming FPGAs** via the BlueBridge application. For this purpose, the *HPT* (HW Programmer & Tester) -a generic FPGA programmer board (using a UART I/F) developed by Dionysios Efstathiou [25]- has been called upon which utilizes the *ReRun* instruction set and API created by Thomas Kyriakidis [26]. The whole venture has been unsuccessful though, due to the low baud rates (1200 bps) supported by the BlueBridge application (for reasons explained below). The HPT system could not be made to operate at such a low speed. Either way, the idea of remotely programming FPGAs is original, for one thing; in future versions of the BlueBridge and mainly of the Bluetooth Module at hand, this task will be some trivial problem to solve.

As mentioned above, an issue concerning the BlueBridge application is its low baud rate support. The BlueApplE communicates with the Bluetooth Module (over the HCI) at maximum speed of 115.2 kbps. However, it has been shown that for every "useful" byte of transmitted information through the BlueBridge, a 10-byte ACL data packet is transferred, effectively reducing the system throughput to

1/10 of 115.2 kbps, i.e. 11520 bps, further falling to 9600 baud which is the closest standard frequency (fig. 7.1).

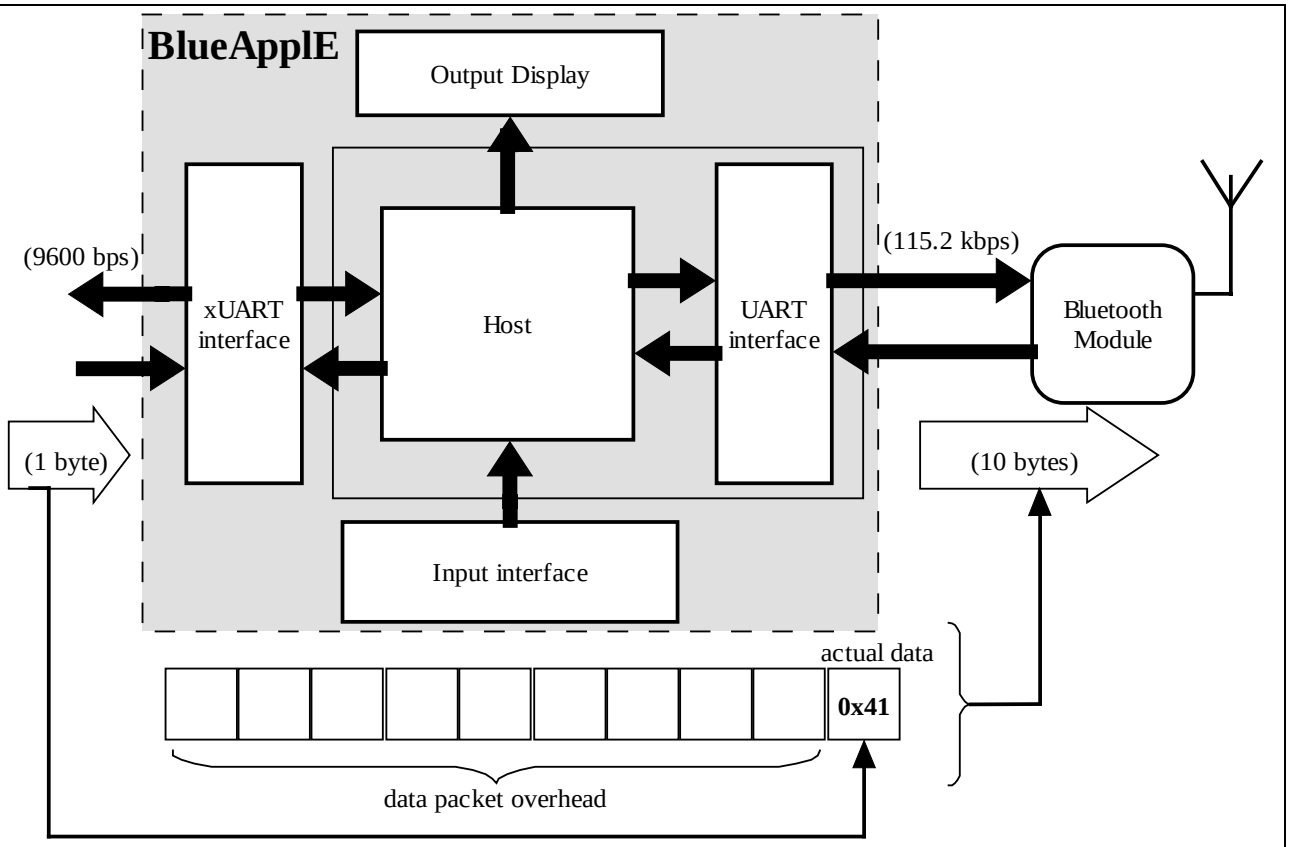


Fig. 7.1: BlueBridge UART speeds

Yet, the system does not manage to work even at this speed and tests have shown that bottleneck disappears when transmitting from the external application at only 1200 baud! This is indeed a low data rate but the BlueApplE system cannot be further boosted for the following reasons:

1. the maximum supported baud rate for the AVR built-in UART (used as the HCI transport) is 115.2 kbps, which is the maximum selected. This limitation thwarts the Bluetooth Module which supports UART speeds of up to 460.8 kbps, i.e. 4 times as fast as the AVR UART.
2. As far as the wastage of the air channel throughput is concerned, it can be dealt with by using larger ACL data packets of up to 672 payload bytes. Even though this would cause the throughput to skyrocket, it is not currently feasible also due to limitations in the AVR: almost **all** 512 bytes of the internal SRAM have been used for implementing the BlueApplE thus making buffering required for large data packet transmission simply *impossible*. The extra deviation to 1200 bps from the theoretically expected 9600 bps

is explained by the fact that the over-the-air transmission presents **great packet loss** drastically reducing the actual data rate. Yet, network measurements have not been conducted (e.g. BER, interference, noise etc.) to figure the exact cause due to development time limitations; moreover, they are outside the thesis scope.

3. in conjunction with the previous reason, the Bluetooth Module further limits BlueBridge speed due to its limited abilities. As seen in chapter 4, it supports neither multi-slot transmissions nor quality-driven data rates (QoS). For this reason, only best-effort and no guaranteed communication is supported. This leaves little space for configuring or pushing the BlueBridge design further.

Even so, the goal of the application which has been the wireless serial communication between devices, has indeed been achieved in a slow but seamless, error-free and user-friendly manner.

7.2 Project debugging

For building and debugging the BlueApplE and BlueBridge systems, the Atmel STK200 development kit has been initially used. This kit has proved very handy in providing a preconstructed platform for developing the design without having to worry with issues such as button debouncing, worn or cut wires, chip pin-outs or any other potential electrical problem. It should be noted though that the on-board crystal of the STK200 is a 4 MHz crystal which has proven unable to support high baud rates (57.6 kbps and further) by cropping the bit trains and putting "noise" on the UART line. It has therefore been replaced with an 11.059200 MHz crystal which gives precise divisions with the standard baud rates and in this case provided smooth and error-free UART operation.

In order to perform fast debugging of the design, one COM port has been used. It has been connected to the HCI transport, i.e. the UART lines connecting the Host with the Bluetooth Module. The serial cable used has been included for **monitoring** the channel, i.e. only the RX and GND signals have been used, with the RX signal connecting to either the TxD or the RxD line of the Host depending on our wish to monitor the HCI commands issued or the events received, respectively. The other end of the serial cable has been connected to the PC and to a *terminal* application that can display ASCII characters in hexadecimal form (fig. 7.2). In this way, all packet traffic could be supervised at any time during system operation.

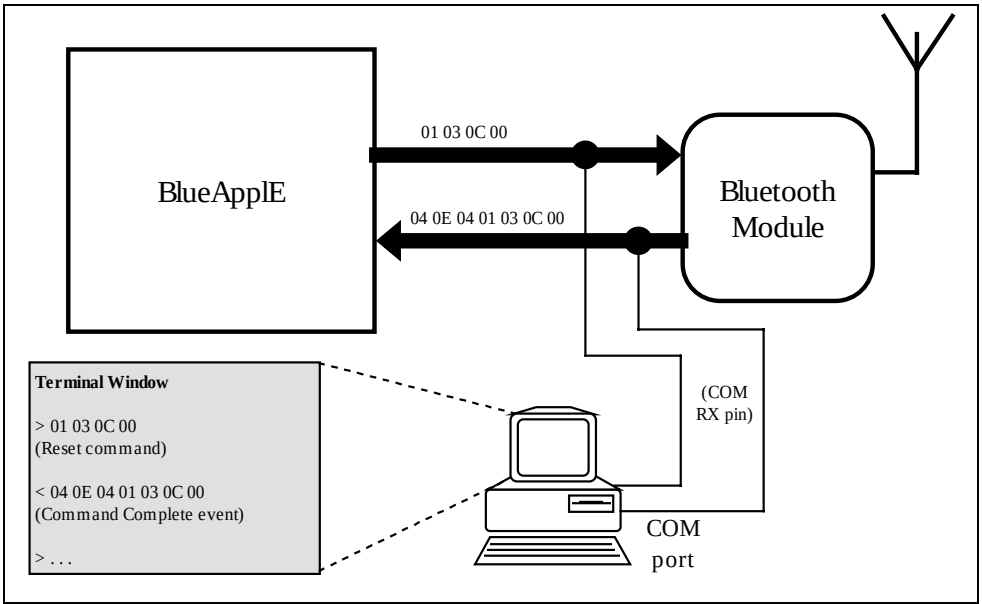


Fig. 7.2: BlueApplE debugging method

Using the same terminal application, the desired functionality of the xUART has been achieved by transmitting data from the terminal application to the xUART and having it **echo** them back to it. Evidently, the terminal application has played the role of the external high-level application widely met in the previous chapter. Figure 7.3 depicts an instance of the terminal having just transmitted a small text file to the xUART (by choosing the "Send File" function).

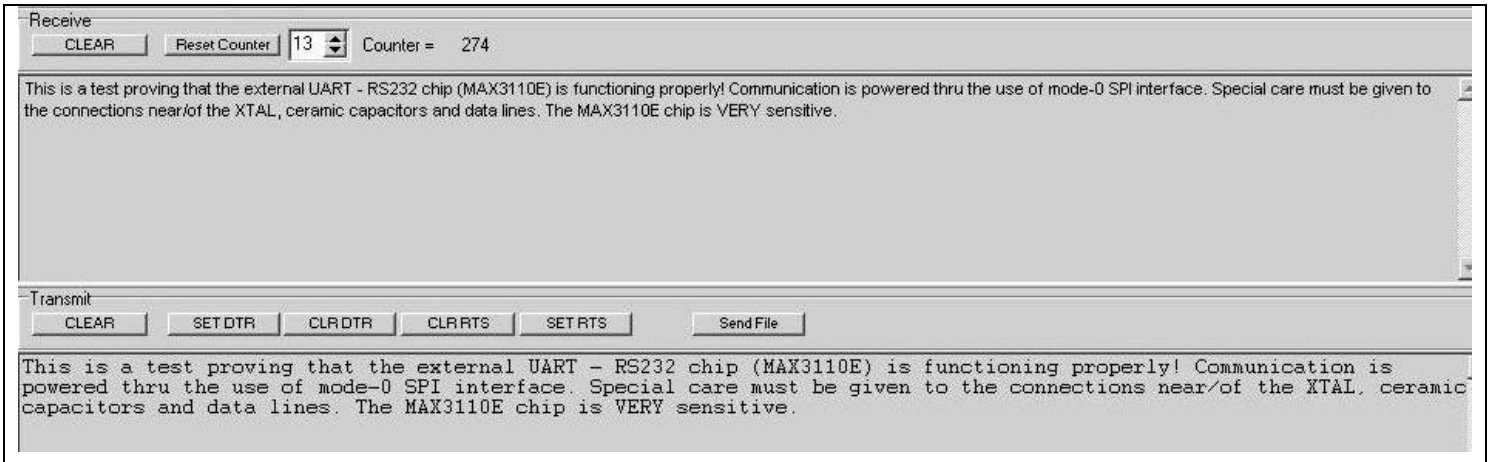


Fig. 7.3: xUART and BlueBridge verification; terminal view

The terminal application has also been used for achieving full BlueBridge operation. From the local terminal a file is sent over to the BlueApplE and it is received on the remote terminal in the same

manner as the one shown in fig. 7.3. Additional verification of the BlueBridge has been achieved by programming an FPGA, as previously noted.

7.2 PCB design and design cost

While the STK200 -Atmel development kit issued by Kanda- has been used in the beginning, later on the full design has been implemented on a breadboard since the STK200 kit became more and more difficult to handle as more components were added to the design (e.g. multiple bus lines and header connectors had to be used). Thus, the STK200 has been used only for programming the AVRs.

The final step in system implementation came with the design of two versions of a PCB -named "Bluetooth Host & Applications Board"- which incorporates *both* the BlueApplE and the BlueBridge systems. The first one has been developed as a **generic** applications board since all unused AVR pins as well as those pins having a temporary nature, e.g. the Output Display, AVR_on pin (PB0), the command selection pins (PD5, PD4) etc. were deliberately left off-board. This PCB is depicted in figure 7.4.

All the above mentioned pins are directed to header connectors and -from there- to a separate breadboard or daughterboard containing LEDs for making these pins operational (fig. 7.5). As it can be extracted from the table of fig. 5.3, all crucial BlueApplE functions have been squeezed to fit in **only** two of the ports of the AVR; port B and port D. Actually, also half of port D is also driven off-board except for the pins PD0 (RX), PD1 (TX), PD2 (INT0) and PD3 (INT1) used for the built-in chip and the interrupts respectively. The remaining pins of port D are secondarily bound for attaching external SRAM modules to the AVR. Currently these pins are used for driving system status LEDs but if the Output Display (which is also driven off-board) is implemented in a different way (e.g. an LCD is used), all port A, port C and the high four pins of port D can be freed. This is the rationale behind the first PCB, i.e. to provide a development board for advanced versions of the BlueApplE system.

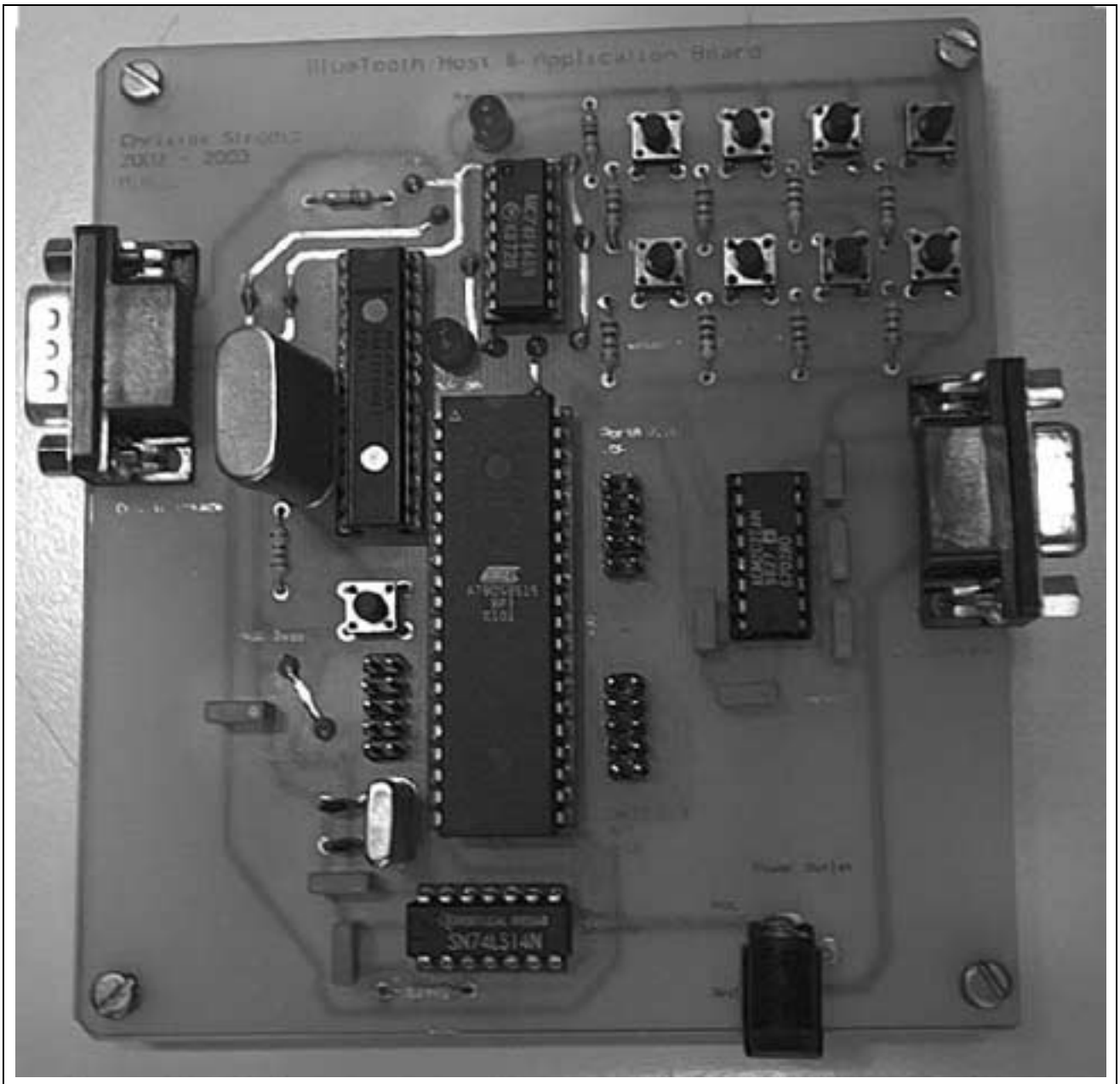


Fig. 7.4: PCB top view, version 1

(Left header connector: PD7..PD4, Vcc, Gnd

Top-right header connector: PA7..PA0

Bottom-right header connector: PC7..PC0)

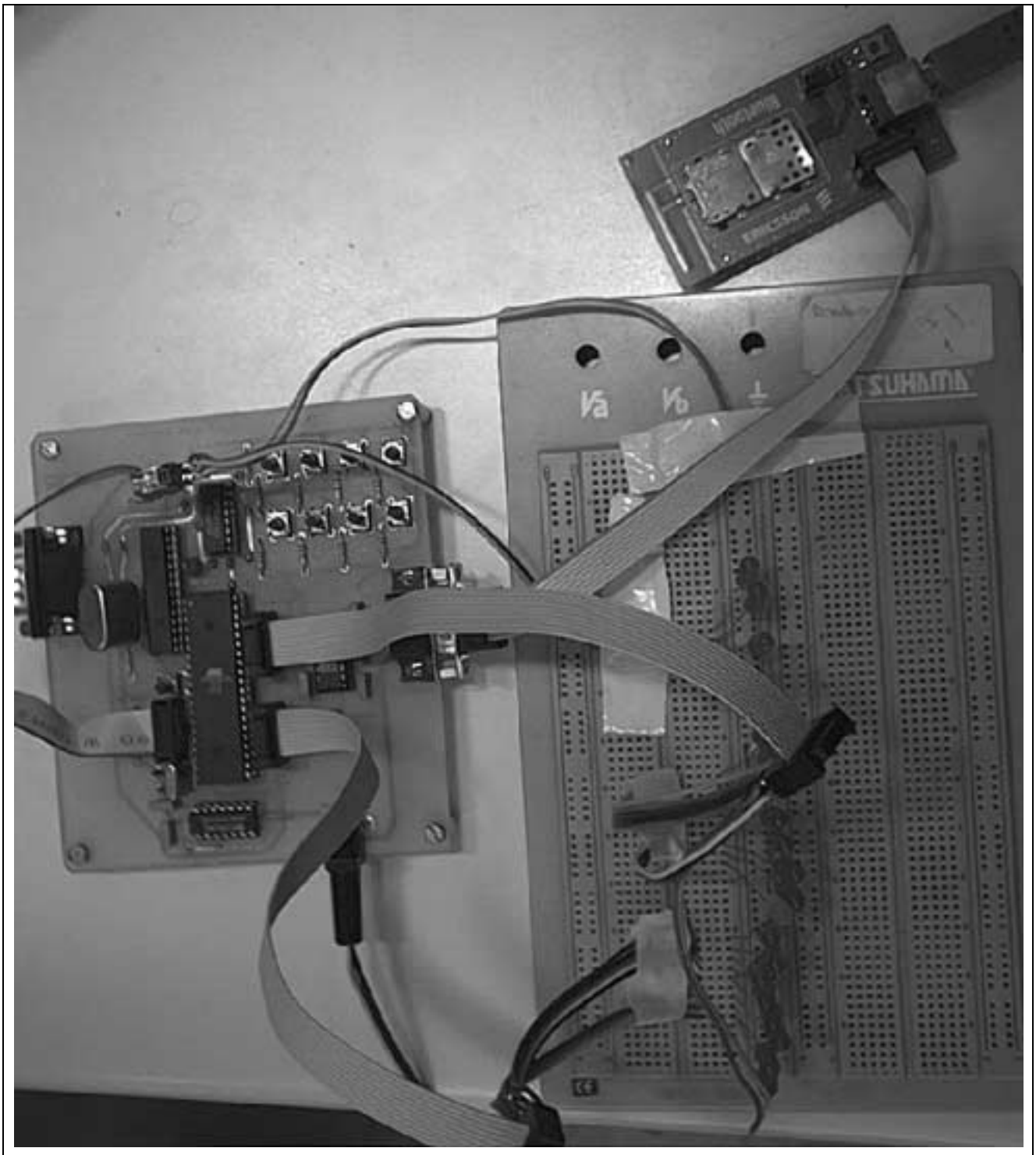


Fig. 7.5: PCB v.1 with external circuitry

The second version of the PCB in fact is a **full instance** of the complete design implemented in this thesis. It incorporates all HW modules (i.e. the Output Display and system status LEDs) as well as the Bluetooth Module itself in order to provide a *solid* design and also an overview of all the parts used. This PCB is depicted in figure 7.6 and includes no header connectors except for the one used to drive the Bluetooth Module. The Module is mounted on the PCB and receives power and internal signaling by the PCB itself[57] (through the above-mentioned header). The header connector cannot be omitted since the Module provides no other means of I/O (e.g. PCI). Should new versions of the above PCBs be produced, a specific point must be stressed out. As seen in the above figures, the SPI lines between the AVR and the MAX3110E are very sensitive and therefore must be as thick and as short as possible. Actually, the whole PCB has been designed *around* those connections so as to avoid violating the above restrictions. Added care must also be paid to the UART and xUART lines (RX and TX) which must also be as short as possible and to present as smooth routing lines as possible (i.e. 90 degree turns should be avoided). By respecting the above issues, system performance is greatly enhanced.

Apart from that, the PCB construction has allowed for an estimation of the system overall cost, as shown in figure 7.7. A total of *approximately* 11.87 € is needed for the components of this system. This indeed proves the low-cost nature of the design, one of the key elements set in chapter 1.

	Component	Price (by approx.)
	Atmel AVR 90S8515	6.00 €
	Maxim-IC MAX3110E	2.37 €
	74LS14	3.50 €
	74F148	
	ICL232	
	UART male connector	
	UART female connector	
	Push buttons (9x)	
	Power outlet connector	
	MKT 100nF capacitor (7x)	
	Ceramic 22pF capacitor (4x)	
	Resistor (12x)	
	LED (22x)	
+	Header connector	
	TOTAL	**11.87 €**

Fig. 7.7: BlueApplE & BlueBridge bill

[57] It is noted that the PCB v.1 can also provide power lines for the Bluetooth Module since it exports Vcc and Gnd signals through one of its header connectors (the one used by port D). Yet, separate cables have to be drawn from the PCB to the Module.

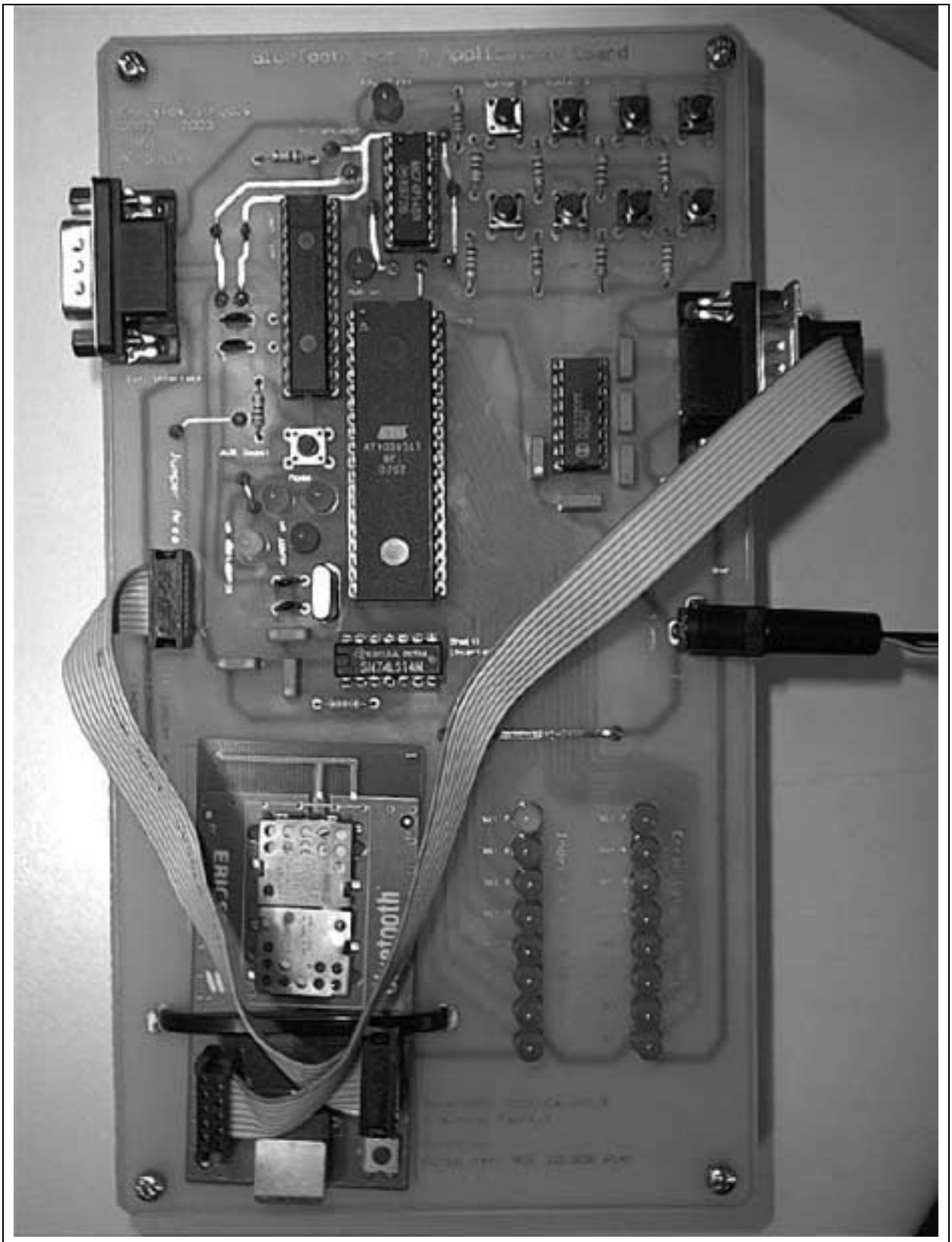


Fig. 7.6: PCB top view, version 2

7.3 Power consumption

Another key element set in chapter 1 has been low power consumption. By measuring the average power consumption of the design -that is, with no LEDs on- a value of **65 to 80 mA** is obtained (no Bluetooth Module current is included). This deviation is caused by environmental variations in temperature as well as LED switching operation. Such *low* current values are vastly stemming from the following reasons:

- SW-based: The Analog Comparator included in the AVR is powered down by setting a specific bit (ACD) in the ACSR controlling its function. In the 5V-range of operation, this effectively reduces power consumption by 0.42 mA. Also, when the AVR does not perform any operation, instead of performing *busy waiting* (e.g. executing an endless loop), it enters the **idle mode** of operation in which it remains responsive, i.e. sensitive to external and internal interrupts, while at the same time cutting down on power consumption by approx. 9 mA (with a 11.0592 MHz XTAL). This mode is entered by executing a "sleep" command while in the endless loop, which causes the CPU to go offline.
- HW-based: On the HW level, the most crucial component -in terms of power consumption- is the MAX3110E. As discussed in chapter 6, this chip demonstrates remarkably low current requirements. Using typical values for the application such as temperature $T_A = 25^0$ Celsius and baud rate B = 115.2 kbps, a supply current of approximately 270 μA is required.

8

Conclusions & Future Work

8.1 Conclusions

In this thesis, a review of Bluetooth technology has been detailed and a development system has been built and tested. The main points of its protocol stack have been investigated and important emanating features have been explained. In an attempt to harness this new wireless technology, a **Blue**tooth **Appl**ications **E**nvironment has been developed which substantiates a Host platform for controlling an off-the-shelf Bluetooth Module. Great effort has been put at making this platform **generic**, i.e. it is designed to support any kind of Bluetooth Module (the HCI transport being the UART I/F) that is Bluetooth-Qualified and complies with the Bluetooth Specification v1.1. Even though the Module at hand is of limited functionality, 3000+ lines of assembly code make this platform capable of connecting to multiple devices and of making provisions for SCO links (although not implemented in this version), to name a few. The platform is also **fully symmetric** in the sense that assembly code is developed in a uniform manner for both master and slave devices; no modifications or special configuration is needed for assuming different roles. Even though the platform provides *direct* access to the HCI layer (with no abstraction layers), strenuous or tedious tasks have been **automated** for making life for the external user easier. Functionality is backed up with a plethora of indication LEDs which inform of system status, error or warning.

Apart from the basic platform (BlueApplE), an application has also been developed, a **Blue**tooth Data **Bridge** effectively implementing a wireless UART cable. Although the communication speed is low due to specific technical reasons, the application is an excellent showcase for displaying part of the Bluetooth technology potential. Even though, the BlueBridge has been unsuitable for remotely downloading a bit-file to an FPGA due to its low speed, a new field of Bluetooth application is being suggested. All design has been subject to **low-cost** and **low-power** features as well as **portability**,

reconfigurability and **modularity**. Those last elements make it easy to upgrade, in ways discussed in the next section.

8.2 Future work

The design is far from being saturated. Temporal and financial constraints have limited development to the instance presented in this document. The level of completion achieved is more than satisfactory but additions and modifications to the design can easily be made, as follows:

1. more HCI commands can be included in the design. While the most essential commands have been implemented already, additional commands will provide more sophisticated and effective handling of the Bluetooth Module. Also, higher interaction with the various HCI command parameter fields can be provided. Needless to say, along with new commands, new events may also need to be included (in the way shown in chapter 5), followed by appropriate decoding of their fields.
2. support for voice (SCO packets) can be introduced. Relative structure is included in the Host but must be further deployed to allow for SCO packet issue and reception. Apart from that, an external A/D converter and an encoder is required for acquiring digital voice which will -of course- constitute the payload of the SCO packets.
3. another AVR can be used instead of the AT90S8515. Some tests have been run under the **ATmega161** which is far fresher a version than the AT90S8515. It has many new features but its key ones for this project are: i) its ability to store and load data in the program memory (16 Kbytes Flash) even during normal execution (instead of only loading data), ii) its increased 1 KByte of internal SRAM (instead of only 512 Bytes), and iii) its *dual* built-in UARTs. This chip is somewhat more expensive than the AT90S8515 one but the first two elements cited, make the available system memory practically *unlimited*, thus lifting the constraint of few HCI commands and of ACL data packets of 1-byte payload (no buffering). By increasing the number of payload bytes, system throughput will skyrocket and BlueBridge data rate will rise vertically. Also, the dual-UART feature lifts the constraint of using an external UART as the MAX3110E chip, thus releasing all SPI pins used for communication between the AVR and the MAX3110E and also the INT1 pin used for incoming data from it. Last but not least, the PCBs seen in chapter 7 have been designed to **support** AT90S8515 and ATmega161 alike.
4. apart from using a newer version of the AVR, external SRAM memory can be attached to the existing AVR for solving the above-mentioned problems (due to the limited memory). As explained in chapter 7, pins PD7 (/RD), PD6 (/WR), ICP and ALE can be used for driving an external memory chip of up to

64 Kbytes size (a latch needs to be included in this case). This upgrade can be easily instantiated on the PCB v.1 which provides ready-to-use header connector pins for the above-mentioned AVR pins.

5. more sophisticated error handling patterns than the one used (with the OC1A compare register of TC1) can be used to catch all faulty cases by performing some smart algorithm on the received events. In case the existing one is used, support for more UART baud rates (currently only for 57.6 kbps) should be added through some dynamically performed calculation or -simply- through the use of a LUT holding values for all baud rates.
6. the Output Display can be thrown away as well as the rest status LEDs and a LCD can be used instead. Judging from the 256x256-dimensioned LCD by Seiko, approximately 10 pins are reserved for LCD operation along with some external circuitry (e.g. trimmer for luminosity etc.), leaving more than 10 AVR pins free. An added advantage in this case is that user-friendliness skyrockets; the drawback is that LCD screens are -generally speaking- very expensive and quite power demanding.
7. even further, the Input Interface can be substituted by a more compact or more handy I/F. Should a LCD be used (as in the above case), the Input Interface can be combined with it making the use of less buttons possible.
8. a USB core can be written or acquired by some external provider (e.g. the USB core developed by Atmel) in order to implement the HCI layer on the USB transport layer. Data rates will also skyrocket in this case, since the USB I/F offers rates of 1 Mbps which is definitely no match for the 115.2 kbps of the UART I/F.

Appendix A

RADIO	Coverage area (Transmit power):	• up to 10 meters (33 ft.) with 1mW (0dBm) solution • up to 100 meters (328 ft.) with 100 mW (20dBm) solution
	Frequency band	• 2.4 GHz - the unlicensed ISM band (83.5 MHz, divided into 79 RF channels 1 MHz apart, is available)
	Modulation	• Shaped, binary frequency modulation (Gaussian Frequency Shift Keying) • BT = 0.5 • Modulation index = 0.28 <h< 0.35
	Receiver sensitivity	• -70dBm at 0.1% Bit Error Rate
	Physical channel	• a pseudo-random hopping sequence hopping through the 79 RF channels. • 1600 hops/s gives 625µs long Time Slots for packet transmission
	Symbol rate	• 1Ms/s
TOPOLOGY	A Piconet is formed when one device (A) sends an Inquiry, and another device (B) answers. The first device (A) now Page the second (B) and establish a Physical Link. In this Piconet "A" is the Master and "B" is the slave. One Master can have up to seven active slaves. Slaves can participate in different Piconets and a master of one Piconet can be the slave in another. This is known as a Scatternet. Up to 10 piconets within range can form a Scatternet without noticeable performance degradation.	
PYSICAL LINKS	Synchronous Connection-Oriented (SCO) link	• circuit switching • symmetric, synchronous services • slot reservation at fixed intervals
	Asynchronous Connection-Less (ACL) link	• packet switching • (a)symmetric, asynchronous services • polling access scheme
	A: up to three simultaneous synchronous voice channels, or a channel which simultaneously supports asynchronous data and synchronous voice. • each voice channel supports a 64 kb/s synchronous (voice) channel in each direction. B: an asynchronous data channel, • the asynchronous channel can support maximal 723.2 kb/s asymmetric (and still up to 57.6 kb/s in the return direction), or 433.9 kb/s symmetric. • a Master can share an asynchronous channel with up to 7 simultaneously active slaves in a Piconet. • by swapping active and parked slaves out respectively in the piconet, 255 slaves can be virtually connected using the PM_ADDR (a device can participate again within 2 ms). • to park even more slaves the BD_ADDR can be used. There is no limitation to the number of slaves that can be parked.	
ADDRESS-ING	Bluetooth Device Address (BD_ADDR)	• 48-bit IEEE 802 address
	Active Member Address (AM_ADDR)	• 3-bit Active Member slave address • all-zero broadcast address
	Parked Member Address (PM_ADDR)	• 8-bit Parked Member slave address
ERROR CORRECTION	Forward-Error Correction (FEC)	• 1/3 rate: bit-repeat code • 2/3 rate: (15,10) shortened Hamming code
	Automatic Repeat Request (ARQ)	• 1-bit fast ACK/NAK • 1-bit sequence number • header piggy-backing • retransmitted on another frequency

SECURITY	Authentication	• challenge/response system with E1 algorithm
	Encryption (privacy)	• encrypts data between two devices • stream cipher with E0 algorithm
	Key management	• configurable encryption key length (16 bytes)
	Initialization	• by user interaction
POWER CONSUM-PTION	Standby current	• < 0.3 mA (3 months)
	Voice mode	• 8-30 mA (75 hours)
	Data mode average	• 5 mA [0.3-30mA, 20 kbps, 25%] (120 hours)
	Hold & Park modes	• 60 μA
PROTOCOL STACK		| Applications like e.g. WAP, vCard, vCal | | AT Commands | OBEX | TCP/IP | | Telephony Control Specification (TCS) | RFCOMM | (Service Discovery Protocol) SDP | | Logical Link Control and Adaptation Protocol (L2CAP) | | Audio | Link Manager (LM) | | Baseband | | Bluetooth Radio |
PRO-FILES	The following 13 Profiles are described in Bluetooth Specification v1.0B: Generic Access Profile, Service Discovery Application Profile, Cordless Telephony Profile, Intercom Profile, Serial Port Profile, Headset Profile, Dial-up Networking Profile, Fax Profile, LAN Access Profile, Generic Object Exchange Profile, Object Push Profile, File Transfer Profile and Synchronization Profile	

Fig. A.1: General Bluetooth features [12]

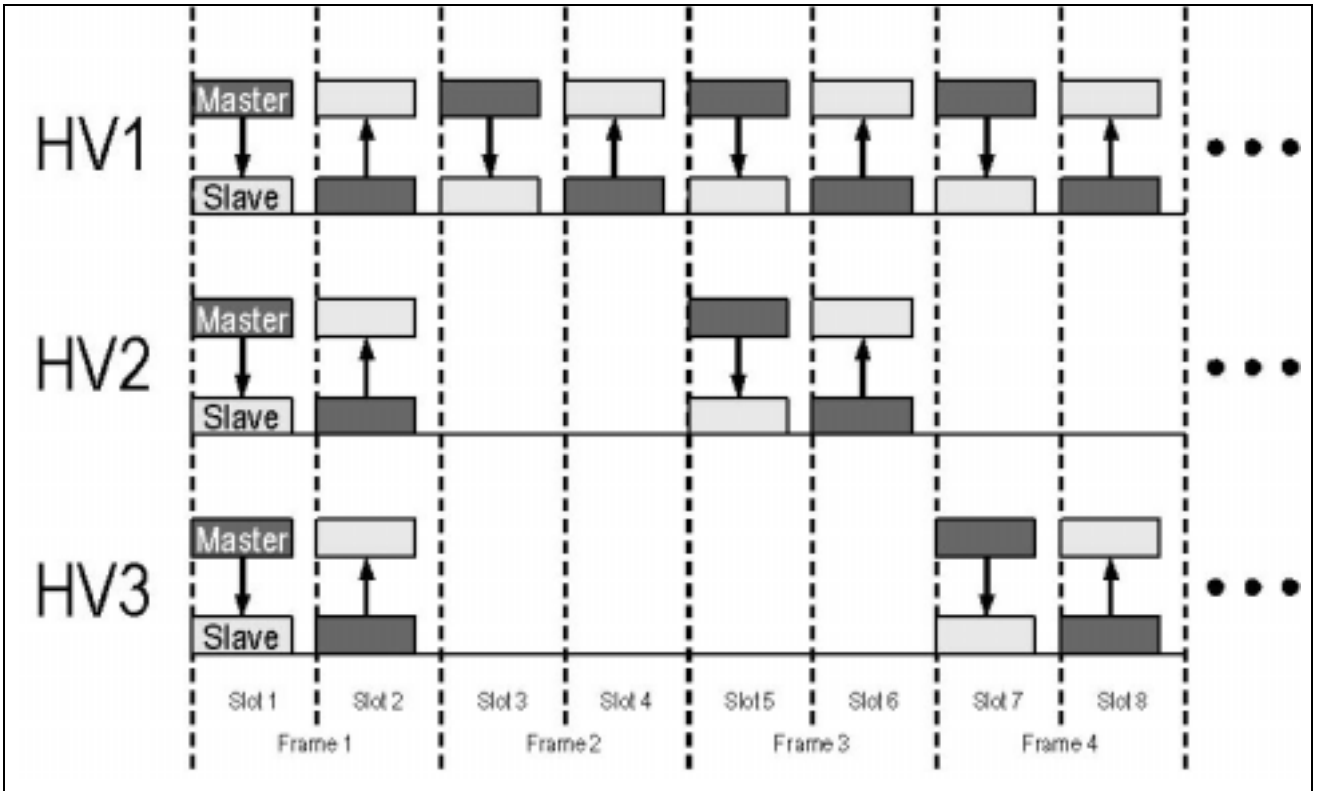


Fig. A.2: SCO link type: HV(1|2|3) packets

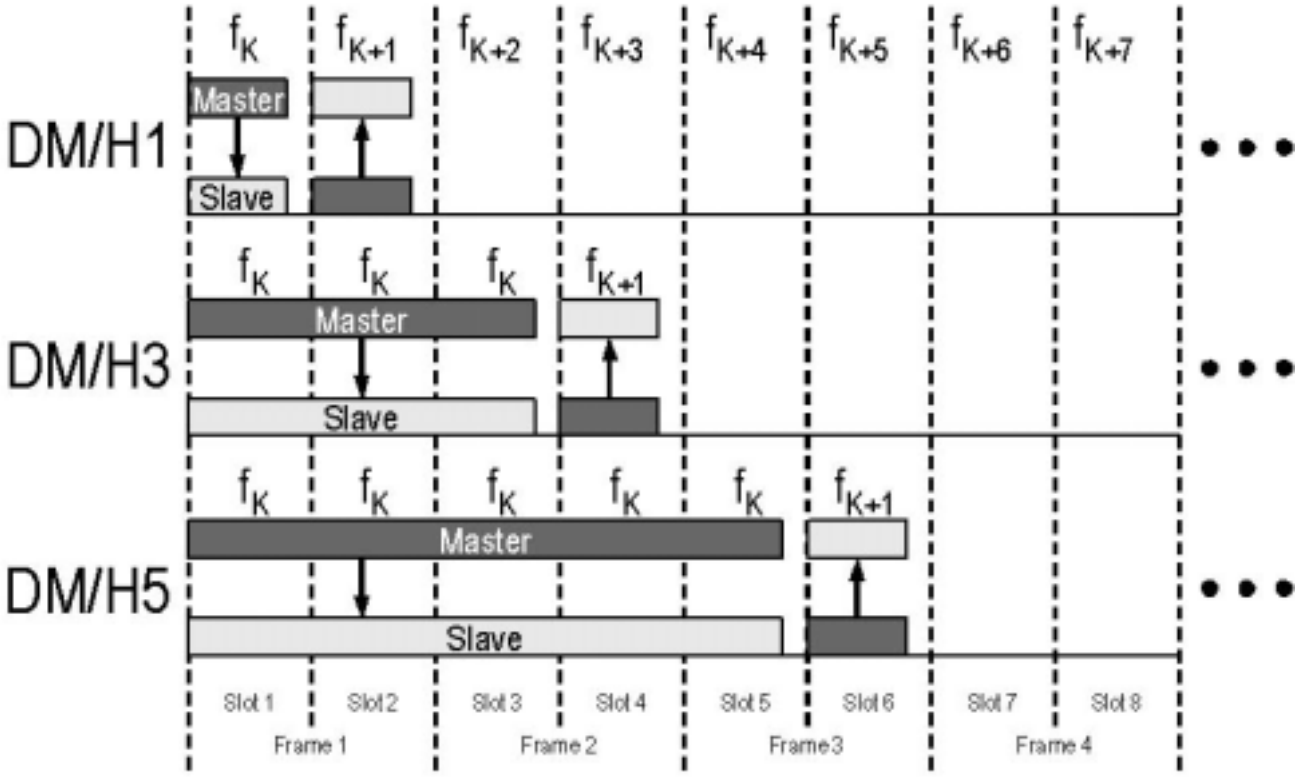


Fig. A.3: ACL link type: D(M|H)(1|3|5) packets

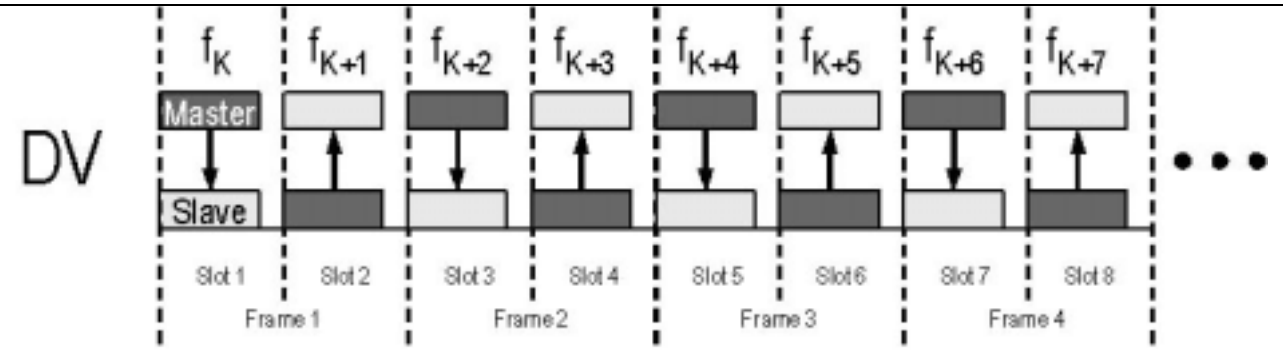


Fig. A.: ACL- / SCO- hybrid DV packet

Standard	Bluetooth	HomeRF	IEEE 802.11	IrDA
Frequency Wavelength	2.4 GHz	2.4 GHz	2.4 GHz	high frequency - over 1014 Hz
Peak Data Rate	1 Mbps	1.6Mbps	2 Mbps	16Mbps
Security Measures	- Unique public address - Two secret keys - A random number different for each transaction	- 56-bit encryption - Frequency-hopping Spread Spectrum - 24-bit network IP	- WEP (Wireless Equivalency Protocol) - Direct Sequence Spread Spectrum	N/A
Optimum Operating Range	<10m	50 m	50 m indoors, 100 m outdoors	<2 m
Voice Network Support	Via IP and Cellular	Via IP and PSTN	Via IP	Via IP
Data Network Support	Via PPP	TCP/IP	TCP/IP	Via PPP
Best suited for a specific purpose or device type	- Phone hands-free headset - Stereo Headphones - Laptops - PDA Devices	- Laptops - Gateways - Cable modems with wireless gateways built in	- Laptops - Desktops - PDAs	- Laptops - Desktops - Mobile Phones - Printers...
Focus	- Mobility - Cellular Connectivity - Cost	- Simplicity - Cost - Voice/Scalability	- Performance - Roaming - Security	- Simplicity
Target Market	Mobile Market	Home Market	Business Market	N/A
Bluetooth Use Compared	N/A	Access Networking	Access Networking	Cable Replacement
Target Chip / Transceiver Cost (estimated)	$5	$20	$70	$1
Relative Cost	Medium	Medium	Medium	Low
Advantages	- Low power requirement - Significant industry support - Mobility and connectivity at the low cost	- Lower power requirement than IEEE 802.11 - Higher connection speed than Bluetooth - Lower cost than IEEE 802.11 - Longer range of communication than Bluetooth	- Highest connection speed - Longest communication range	- Existent base of installed IR ports (estimated to be 60 mil) - Lowest Cost - Lowest Power requirement - No interference with the existing LAN technologies
Disadvantages	- Interference with IEEE.802 which is considered to be the future technology for Wireless LAN	- Interference with IEEE.802 which is considered to be the future technology for Wireless LAN	- Significantly high cost - Not scalable to personal devices	- Not secure - Requires line-of sight for communication between devices - Very short range

Fig. A.1: Major attribute list of various wireless protocols

Appendix B

The profiles have been developed in order to describe how implementations of user models are to be accomplished. The user models describe a number of user scenarios where Bluetooth performs the radio transmission. A profile can be described as a vertical slice through the protocol stack. It defines options in each protocol that are mandatory for the profile. It also defines parameter ranges for each protocol. The profile concept is used to decrease the risk of interoperability problems between different manufacturers' products.

The four general profiles defined; Generic Access Profile (GAP), the Serial Port Profile, the Service Discovery Application Profile (SDAP) and the Generic Object Exchange Profile (GOEP), are the base for all user models and their profiles (fig. B.1).

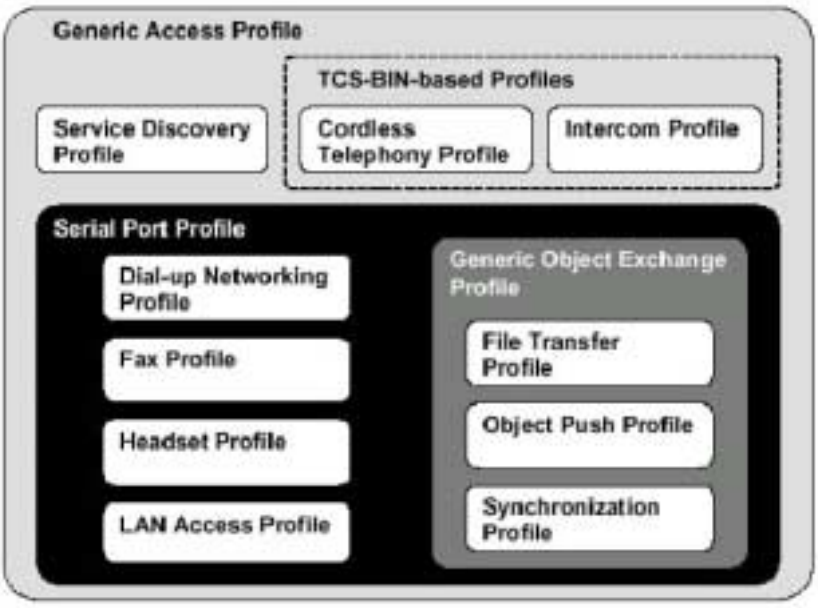


Fig. B.1: Basic Bluetooth profiles

The Bluetooth profile structure and the dependencies of the profiles are depicted above. A profile is dependent upon another profile if it re-uses parts of that profile by implicitly or explicitly referencing it. Dependency is illustrated in the figure: a profile has dependencies on the profile(s) in which it is contained – directly and indirectly. For example, the Object Push profile is dependent on Generic Object Exchange, Serial Port, and Generic Access profiles. Concisely, the profiles are:

- **Generic Access Profile (GAP)** - This profile defines the generic procedures related to discovery of Bluetooth devices (idle mode procedures) and link management aspects of connecting to Bluetooth devices (connecting mode procedures). It also defines procedures related to use of different security levels. Essentially this profile describes how the lower layers (LMP and Baseband) are used, along with some higher layers. This profile:
    - States the requirements on names, values and coding schemes used for names of parameters and procedures experienced on the user interface level.
    - Defines modes of operation that are not service- or profile-specific, but generic to all profiles.
    - Defines the general procedures that can be used for discovering identities, names and basic capabilities of other Bluetooth devices that are in a mode where they can be discoverable. Only procedures where no channel or connection establishment is used are specified.
    - Defines the general procedure for how to create bonds between Bluetooth devices.
    - Describes the general procedures that can be used for establishing connections to other Bluetooth devices
- **Service Discovery Profile (SDP)** - The service discovery profile defines the protocols and procedures that shall be used by a service discovery application on a device to locate services in other Bluetooth-enabled devices using the Bluetooth Service Discovery Protocol (SDP). With regard to this profile, the service discovery application is a specific user-initiated application. In this aspect, this profile is in contrast to other profiles where service discovery interactions between two SDP entities in two Bluetooth-enabled devices result from the need to enable a particular transport service (e.g. RFCOMM, etc.), or a particular usage scenario (e.g. file transfer, cordless telephony, LAN AP, etc., as seen below) over these two devices. Service discovery interactions of the latter kind can be found within the appropriate Bluetooth usage scenario profile documents. The main purpose of this profile is to describe the use of the lower layers of the Bluetooth protocol stack (LC and LMP). To describe security related alternatives, also higher layers (L2CAP, RFCOMM and OBEX) are included.
- **Cordless Telephony Profile (CTP)** - This profile defines the features and procedures that are required for interoperability between different units active in the '3-in-1 phone' use case. The '3-in-1 phone' is a

solution for providing an extra mode of operation to cellular phones, using Bluetooth as a short-range bearer for accessing fixed network telephony services via a base station. However, the 3-in-1 phone use case can also be applied generally for wireless telephony in a residential or small office environment, for example for cordless-only telephony or cordless telephony services in a PC – hence the profile name 'Cordless Telephony'.

- **Intercom Profile (IP)** - This profile defines the requirements for Bluetooth devices necessary for the support of the intercom functionality within the 3-in-1 phone use case. The requirements are expressed in terms of end-user services, and by defining the features and procedures that are required for interoperability between Bluetooth devices in the 3-in-1 phone use case. More popularly, this is often referred to as the 'walkie-talkie' usage of Bluetooth.
- **Serial Port Profile (SPP)** - The Serial Port Profile defines the requirements for Bluetooth devices necessary for setting up emulated serial cable connections using RFCOMM between two peer devices. The requirements are expressed in terms of services provided to applications, and by defining the features and procedures that are required for interoperability between Bluetooth devices. Essentially, the Serial Port Profile defines the protocols and procedures that shall be used by devices using Bluetooth for RS232 (or similar) serial cable emulation. The scenario covered by this profile deals with legacy applications using Bluetooth as a cable replacement, through a virtual serial port abstraction (which in itself is operating system-dependent).
- **Headset Profile (HP)** - The Headset profile defines the requirements for Bluetooth devices necessary to support the Headset use case. Essentially the Headset profile defines the protocols and procedures that shall be used by devices implementing the usage model called 'Ultimate Headset'. The most common examples of such devices are headsets, personal computers, and cellular phones.
- **Dial-up Networking Profile (DNP)** - The Dial-up Networking profile defines the requirements for Bluetooth devices necessary to support the Dial-up networking use case. Essentially the Headset profile defines the protocols and procedures that shall be used by devices implementing the usage model called 'Internet Bridge'. The most common examples of such devices are modems and cellular phones. Two main scenarios are implemented: the Usage of a cellular phone or modem by a computer as a wireless modem for connecting to a dial-up internet access server, or using other dial-up services, and Usage of a cellular phone or modem by a computer to receive data calls.
- **Fax Profile (FP)** - The Fax profile defines the requirements for Bluetooth devices necessary to support the Fax use case. Essentially the Fax profile defines the protocols and procedures that shall be used by devices implementing the fax part of the usage model called 'Data Access Points, Wide Area Networks'. A Bluetooth cellular phone or modem may be used by a computer as a wireless fax modem to send or receive a fax message.

- **LAN Access Profile (LAP)** - The LAN Access Profile for Bluetooth devices consists of 2 parts. Firstly, this profile defines how Bluetooth-enabled devices can access the services of a LAN using PPP. Second, this profile shows how the same PPP mechanisms are used to form a network consisting of two Bluetooth-enabled devices. Basically this profile defines LAN Access using PPP over RFCOMM. (There may be other means of LAN Access in the future).
- **Generic Object Exchange Profile (GOEP)** - This profile defines the requirements for Bluetooth devices necessary for the support of the object exchange usage models. The usage model can be, for example, Synchronization, File Transfer, or Object Push model. Essentially, the purpose of this document is to work as a generic profile document for all application profiles using the OBEX protocol.
- **Object Push Profile (OPP)** - This profile defines the requirements for the protocols and procedures that shall be used by the applications providing the object push usage model. The object push usage model makes use of the underlying Generic Object Exchange Profile (GOEP) to define the interoperability requirements for the protocols needed by applications. Typical scenarios covered by this profile are: Object Push, Business Card Pull & Business Card Exchange, all of which involve the pushing/pulling of data objects between Bluetooth devices.
- **File Transfer Profile (FTP)** - This profile defines the requirements for the protocols and procedures that shall be used by the applications providing the file transfer usage model. The file transfer usage model makes use of the underlying Generic Object Exchange Profile (GOEP) to define the interoperability requirements for the protocols needed by applications. Typical scenarios covered by this profile involving a Bluetooth device browsing , transferring and manipulating objects on/with another Bluetooth device.
- **Synchronization Profile (SP)** - This profile defines the requirements for the protocols and procedures that shall be used by the applications providing the synchronization usage model. The synchronization usage model makes use of the underlying Generic Object Exchange Profile (GOEP) to define the interoperability requirements for the protocols needed by applications. Typical scenarios covered by this profile involving a computer instructing a mobile phone or PDA to exchange PIM data , or vice versa (a mobile instructing a computer to exchange PIM data), or automatically starting synchronization when two Bluetooth devices come within range.

Various usage scenarios can be based upon these profiles. The major usage models proposed are as follows:

- **File Transfer** - The File Transfer usage model offers the capability to transfer data objects from one Bluetooth device to another. Files, entire folders, directories and streaming media formats are supported in this usage model. This usage model also offers a possibility to browse the contents of the folders on a remote device. Furthermore, push and exchange operations are covered in this usage model, e.g., business card exchange using the vCard format.
- **Internet Bridge** - The Internet Bridge usage model describes how a mobile phone or cordless modem provides a PC with dial-up networking capabilities without need for physical connection to the PC. This networking scenario needs a two-piece protocol stack, one for AT-commands needed to control the mobile phone and another stack to transfer payload data.
- **LAN Access** - The LAN Access usage model is similar to the Internet Bridge user model. The difference is that the LAN Access usage model does not use the protocols for AT-commands. The usage model describes how data terminals use a LAN access point as a wireless connection to a Local Area Network. When connected, the data terminals operate as if it they were connected to the LAN via dialup networking.
- **Synchronization** - The synchronization usage model provides means for automatic synchronization between for instance a desktop PC, a portable PC, a mobile phone and a notebook. The synchronization requires business card, calendar and task information to be transferred and processed by computers, cellular phones and PDAs utilizing a common protocol and format.
- **Three-in-One Phone** - The Three-in-One Phone usage model describes how a telephone handset may connect to three different service providers. The telephone may act as cordless telephones connecting to the public switched telephone network at home charged at a fixed line charge. This scenario includes making calls via a voice base station, and making direct calls between two terminals via the base station. The telephone can also connect directly to other telephones acting as a "walkie-talkie" or handset extension i.e. no charging. Finally, the telephone may act as a cellular telephone connecting to the cellular infrastructure. The cordless and intercom scenarios use the same protocol stack.
- **Ultimate Headset** - The Ultimate Headset usage model defines how a Bluetooth equipped wireless headset can be connected to act as a remote unit's audio input and output interface. The unit is probably a mobile phone or a PC for audio input and output. As for the Internet Bridge user model, this model requires a two-piece protocol stack; one for AT-commands needed to control the mobile phone and another stack to transfer payload data, i.e. speech. The AT commands controls the telephone regarding for instance answering and terminating calls.

Appendix C

Below, an example of how a HCI command is formed follows. The command used is Read_BD_ADDR which obviously -when issued- returns the BD_ADDR of the local Bluetooth Module. For convenience, the general format of a HCI command packet is repeated in figure C.1.

<table>
<tr><td colspan="8">0 4 8 12 16 20 24 28 31</td></tr>
<tr><td colspan="4">OpCode</td><td colspan="2" rowspan="2">Parameter Total Length</td><td colspan="2" rowspan="2">Parameter 0</td></tr>
<tr><td colspan="2">OCF</td><td colspan="2">OGF</td></tr>
<tr><td colspan="4">Parameter 1</td><td colspan="4">Parameter 2</td></tr>
<tr><td colspan="8">•
•
•</td></tr>
<tr><td colspan="4">Parameter N-1</td><td colspan="4">Parameter N</td></tr>
<tr><td colspan="8">Fig. C.1: HCI command format</td></tr>
</table>

The first thing that needs to be formed is the **OpCode**. For this to happen, the OGF and OCF subfields must be defined. **OCF** equals '0x0009' and **OGF** equals "0x04" since the Read_BD_ADDR command belongs to the *Informational Parameters* group. As far as parameters are concerned, this command has none, so **Parameter Total Length** equals '0x00' (and no parameter bytes follow). So, the unformatted sequence of bytes to be transmitted is as follows:

OCF	**OGF**	**Parameter Total Length**
0x00 0x09	0x04	0x00
Fig. C.2: Packet intermediate form		

The first thing to do is to merge the OCF, OGF fields into the final OpCode field. Since Little-Endian byte order is valid and given that the OpCode is 2 bytes long whereas (OGF+OCF) is 3 bytes long, concatenation and inversion of various bits must take place. This is the crucial point of synthesizing HCI packets. According to the BT Spec. 2 MSBs of the OGF subfield and 6 MSBs of the OCF subfield are thrown away and the remaining bits are merged in the OpCode field. After that, the OpCode needs also to be reversed to assume Little-Endian structure. By following the above steps, the resulting OpCode is: '0x0910' and the Parameter Total Length field ('0x00') follows. Finally, one must not forget that this HCI packet will be transmitted over a specific HCI transport and an additional **packet indicator** field must precede the HCI packet. If the physical transport is the UART, then the indicator used is '0x01' signaling the transmission of a HCI command. The complete packet, ready for transmission is: '0x01' '0x09' '0x10' '0x00'. The above process is graphically shown in figure C.3. Also note that if there were any parameters, then Parameter Total Length would be equal to the *sum of bytes* of those parameters and not equal to their number. Also, Little-Endian pattern would also be followed, e.g. say there are two parameters: '0x1122' and '0x33'; then the transmitted byte order would be: '0x01' '0x09' '0x10' **'0x03' '0x22' '0x11' '0x33'**, i.e. the Little-Endian feature is applied inside every field and not to the whole of the packet.

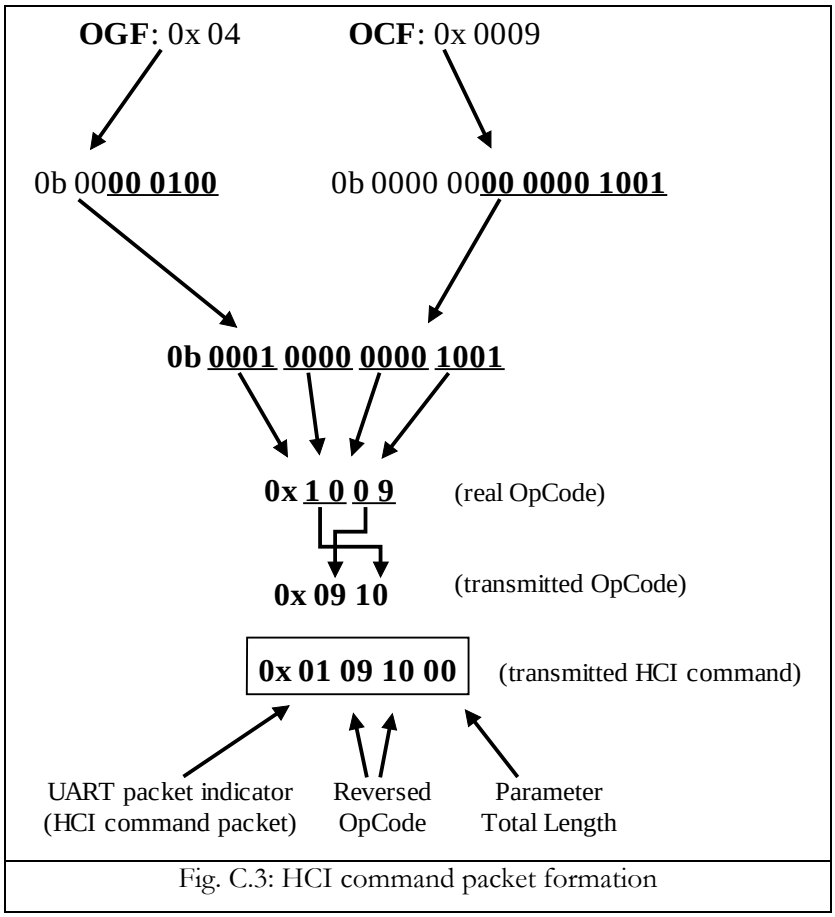


Fig. C.3: HCI command packet formation

Appendix D

The SPI I/F is a very popular way for fast connection between devices with very few wires. Speed in the SPI is the wanted object and in pursuit of speed an eccentric method of writes and reads has arisen, as follows: The SPI assumes a master-slave topology among devices. The master is the device that provides the SPI CLOCK (SCK) to all other devices and is the one that initiates ALL the reads/writes. All SPI-enabled devices contain a shift register. The two shift registers in the master and the slave can be considered as one distributed 16-bit circular shift register (fig. D.1). When data is shifted out from the master to the slave (through MOSI pin), data is also shifted in the opposite direction (through the MISO pin), simultaneously. This means that during one shift cycle, data in the master and the slave are interchanged.

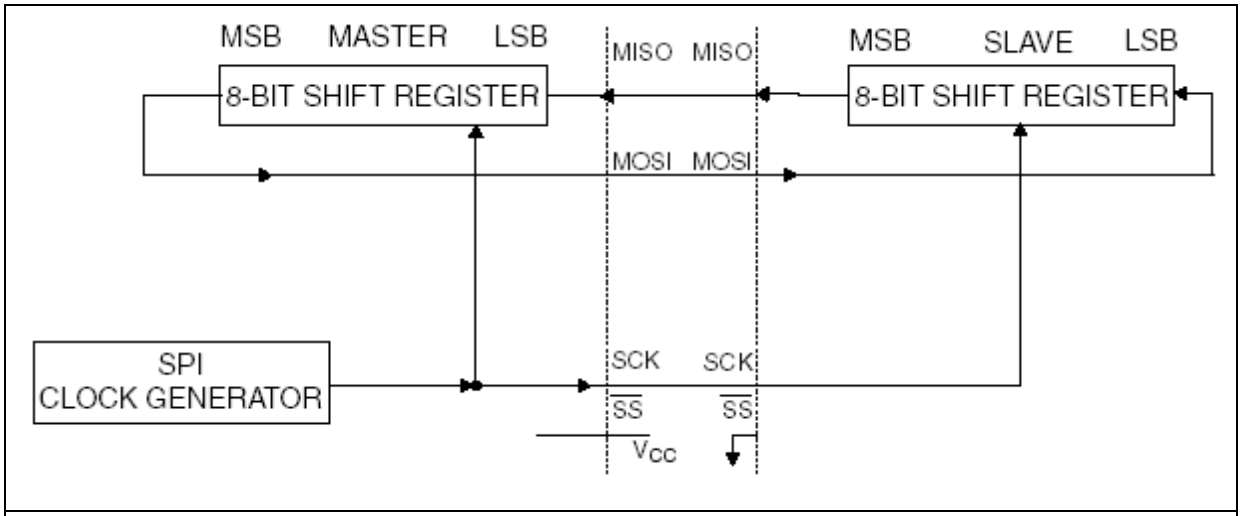


Fig. D.1: SPI master-slave interconnection

The master is always the one which is allowed to initialize a transfer; when the master needs to receive data from the slave, it writes the SPDR with the data for transmission as follows:

```
        out SPDR,temp1 ; Write data to SPDR for transfer
wait :  sbis SPSR,7 ;* Check SPIF for complete transfer; while SPIF != '1', wait.
        rjmp wait   ;*

        in temp1,SPSR ;* Read SPSR and SPDR to: 1) store the received data and 2)
        in temp1,SPDR ;* clear SPIF flag in SPSR. (SPIF is also cleared if the
                      ;* SPI interrupt is triggered).
```

When SPDR has entirely shifted out its data, it will hold the contents of the SPDR of the slave! In case the master wants to read data from a slave, it must also **write** the SPDR in order for valid data of the slave to come over. Since the master wants only to read data, it can as well write trash to the SPDR. Te code segment in this case is as follows:

```
        out SPDR,temp ; Load a random value to SPDR
wait :  sbis SPSR,7 ;* Check SPIF for transfer complete.
        rjmp wait   ;* If busy then wait more.

        in Instruct,SPSR ;* Read SPSR and
        in Instruct,SPDR ;* read SPDR to clear SPIF in SPSR.
```

In the case of the MAX3110E chip, in order to avoid having the master periodically polling the slave for new data (i.e. periodically sending trash to the slave), the chip provides an INT line that is activated when new data are available. Furthermore, in the MAX3110E chip uses 16-bit long words to communicate with the AVR. The solution to the problem is simple: the AVR must transmit 2 bytes each time (a high and a low byte). As seen in the MAX3110E manual, the proper sequence for accessing the chip is to pull the /CS signal low, transmit the 2 bytes and then pull the /CS high. At this end of this process, the master (AVR) will hold the two bytes simultaneously received by the chip. A sample code for the above is as follows:

```
.equ nSS = 4 ; /SS
.equ CONFIGURATION_WORD = 0xC40F
.def temp1 = r16
.def temp2 = r17

cbi PORTB,nSS ; Enable transmission

ldi temp1,high(CONFIGURATION_WORD)
rcall SPI_write

mov temp2,temp1 ;* On next SPI transm. temp1 will be overwritten, so current
                ;* temp1 contents are safe-kept in temp2 for further processing.

ldi temp1,low(CONFIGURATION_WORD)
rcall SPI_write

sbi PORTB,nSS ; Disable transmission
```

In the above code, temp2 is loaded with the data from temp1, because the MAX3110E chip, even when written, provides information back to the master (e.g. more data are available to read etc.). In figure D.2 the sequence of signals during *write configuration* command are depicted.

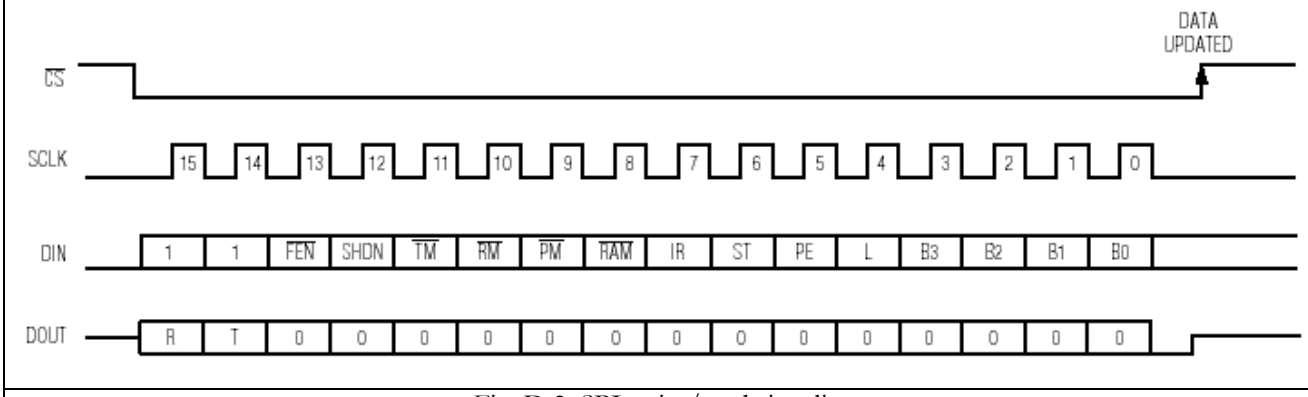


Fig. D.2: SPI write/read signaling

References

Literature

[1] Brent A. Miller, Chatschik Bisdikian, *BT revealed: The insider's guide to an open specification for global wireless Communications*, ISBN: 0-13-090294-2.

[2] David Kammer, Gordon McNutt, Brian Senese, *Bluetooth Application Developer's Guide: The short range interconnect solution*, ISBN: 1-928994-42-3.

[3] Bluetooth SIG, *Specification of the Bluetooth System (specification volume 1), Core.*

[4] Bluetooth SIG, *Specification of the Bluetooth System (specification volume 2), Profiles.*

[5] Ericsson Technology Licensing AB, *Bluetooth Beginner's Guide.*

[6] Riku Mettala, *Bluetooth Whitepaper: Protocol Architecture v1.0.*

[7] Xilinx, *A brief introduction to Bluetooth.*

[8] Ericsson, *Bluetooth: Brand and Qualification.*

[9] Atmel, *Bluetooth Qualification Procedure White Paper.*

[10] Jim Lansford, Ph.D., *Working Towards the Peaceful Coexistence of Wireless PANs, LANs, and WANs.*

[11] N. Golmie, R. E. Van Dyck, and A. Soltanian, *Interference of Bluetooth and IEEE 802.11: Simulation Modeling and Performance Evaluation*, National Institute of Standards and Technology, Gaithersburg, Maryland 20899.

[12] Ericsson, *Bluetooth wireless technology: General Bluetooth Features.*

[13] Ericsson, *Getting Started: Bluetooth Application Tool Kit, LZT 108 4123 R2A.*

[14] Ericsson, *ROK 101 008: Bluetooth PtP Module.*

[15] Ericsson, *Users Manual - Bluetooth PC Reference Stack.*

[16] Atmel, *USB v1.1 Device, 32-bit Embedded Core Peripheral.*

[17] Atmel, *8-bit AVR Microcontroller With 8 Kbytes In-System Programmable Flash: AT90S8515.*

[18] Fairchild Semiconductor, *74F148: 8-Line To 3-Line Priority Encoder.*

[19] Apostolos Dollas, *The Art of Microelectronic Systems*

[20] Ron Mancini, *Examining Switch-Debounce Circuits*, AnalogAngle.

[21] Fairchild Semiconductor, *DM74LS14: Hex Inverter With Schmitt Trigger Inputs.*

[22] Seiko Instrument Inc., *Liquid Crystal Display Module: G121CB1P00C.*

[23] Harris Semiconductor, *ICL232: +5V Powered Dual RS-232 Transmitter/Receiver.*

[24] Maxim-IC, *MAX3110/MAX3111E: SPI/MICROWIRE-Compatible UART and ±15kV ESD-Protected RS-232 Transceivers with Internal Capacitors.*

[25] Dionysios Efstathiou, *Diploma Thesis: Design and Implementation of a Vendor-Independent Universal Programmer for FPGA Technology.*

[26] Thomas Kyriakidis, *Diploma Thesis: Development of a language and Universal Run-Time Environment for FPGA programming.*

[27] Sedra/Smith, *Μικροηλεκτρονικά Κυκλώματα*, Τόμος Α', ISBN: 960-85334-5-7

[28] Sedra/Smith, *Μικροηλεκτρονικά Κυκλώματα*, Τόμος Β', ISBN: 960-7510-10-0

Links

[29] http://www.ericsson.com/bluetooth/ (Ericsson Technology Licensing)

[30] www.comtec.teleca.se (Teleca Comtec homepage and support center)

[31] http://www.bluetooth.com/tech/products.asp (Ericsson partners' full product list)

[32] http://www.clibb.de/ (Bluetooth web traces page)

[33] http://www.infotooth.com/ (palowireless.com Wireless Resource Center)

[34] www.thewirelessdirectory.com

[35] www.kanda.com (Manufacturer and support company of the STK200 kit)

[36] www.atmel.com (Atmel home page)

[37] www.avr-freaks.net (AVR-relative site)

[38] www.maxim-ic.com (Maxim-IC home page)

[39] http://www.xilinx.com/esp/bluetooth/glossary/a.htm (Large Bluetooth glossary)

[40] www.bluetooth.com (Official Bluetooth site)

[41] http://www.forum.nokia.com/main.html (Nokia developer's site)

[42] http://whatis.techtarget.com/ (technical term online dictionary)

Newsgroups

[43] http://groups.yahoo.com/group/bluetech/

[44] http://groups.yahoo.com/group/bluesales/

[45] http://groups.yahoo.com/group/blueinfo/

[46] comp.arch.embedded

[47] comp.lang.vhdl

Product Sheets

[48] Atmel, Bluetooth ISM 2.4 GHz power amplifier.

[49] Atmel, Bluetooth ISM 2.4 GHz front-end IC.

[50] Atmel, Bluetooth ISM 2.4 GHz power amplifier.

[51] Atmel, Single Chip Bluetooth Controller.

[52] Infineon Tech., BlueMoon Single - PMB8760.

[53] Infineon Tech., BlueMoon Single Cellular - PMB8761.

[54] Infineon Tech., BlueMoon UniUSB - PMB8754.

[55] Infineon Tech., BlueMoon UniCellular - PMB8752.

[56] Motorola, 71000 Bluetooth Development Kit.

[57] Motorola, The Bluetooth Platform Solution From Motorola.

[58] CSR, BlueCore2-External, Single Chip Bluetooth System.

[59] Teleca Comtec, CAN-to-Bluetooth Gateway (www.comtec.teleca.se/bluecan.asp).

[60] IBM, BlueDrekar: MW for Bluetooth wireless devices.

[61] IBM, BlueHoc: Bluetooth ad-hoc network simulator.

[62] Synopsys, DesignWare BlueIQ Core.
[63] IAR Systems, Bluetooth Starter Kit.
[64] Teleca Comtec, Ericsson Bluetooth Development Kit.
[65] Teleca Comtec, Ericsson Starter Kit.
[66] Impulsesoft, Impulsesoft Bluetooth Development Kit.
[67] Affix Bluetooth Protocol Stack For Linux (http://affix.sourceforge.net/)
[68] BlueZ - Offical Linux Bluetooth protocol stack (http://bluez.sourceforge.net/)

Glossary

HCI	Host Controller Interface: A physical I/F splitting the Bluetooth protocol stack in two parts. Typically implemented through USB or a UART in Bluetooth components.
HW	Hardware
SW	Software
FW	Firmware
MW	Middleware. In the computer industry, middleware is a general term for any programming that serves to "glue together" or mediate between two separate and often already existing programs.
SIG	Special Interest Group
WLAN	Wireless Local Area Network
PDA	Personal Digital Assistant
ad hoc (network)	An ad-hoc (or "spontaneous") network is a local area network or other small network, especially one with wireless or temporary plug-in connections, in which some of the network devices are part of the network only for the duration of a

	communications session or, in the case of mobile or portable devices, while in some close proximity to the rest of the network.
PAN (or piconet)	Personal Area Network: A term popularized to describe how Bluetooth enables a collection of personal electronic devices to operate together as a logical collective. (The structure created when 2 or more devices create a Master/Slave connection. Each Piconet can have only 1 Master, 1 to 7 active Slaves, and 0-255 Parked Slaves.)
scatternet	A higher level network construct involving 2 or more interconnected Piconets.
HID	Human Interface Device (e.g. mouse, joystick etc.)
BQRB	Bluetooth Qualification Review Board
ISM bands	Industrial Scientific and Medical Bands: FCC allocated RF spectrum in the 900MHz, 2.4GHz, and 5GHz bands. ISM spectrum is freely available for use without a license so long as radiating devices meet basic behavioral guidelines.
FHSS	A spread-spectrum technique in which the carrier frequency jumps or hops into different frequencies with respect to time.
inquiry	A formal method by which Bluetooth devices Discover each other.
page	The method for establishing a formal Piconet connection between a Master and a Slave.
sniff mode	A Connected Mode where an active Slave is granted predetermined, recurring intervals to ignore the Piconet. The Slave can use these periods for any purpose and can rejoin the Piconet with zero latency after any interval.
hold mode	A Connected Mode where an active Slave is granted a predetermined unit of time to ignore the Piconet. The Slave can use this time for any purpose and will rejoin the Piconet with zero latency after T-Hold expires.
park mode	A Mode where a Piconet Slave relinquishes its 3-bit AMA address and is given an 8-bit Parked Member Address (PMA). In this mode the Slave is only obligated to monitor the Piconet at a predetermined beacon interval.
standby mode	The unconnected mode for Bluetooth devices. While in Standby the only Bluetooth obligation is to listen for Inquiries and Pages on an occasional basis.
BD_ADDR	Bluetooth Device Address: A unique 48-bit address for every manufactured Bluetooth device. It is similar to an IEEE 48-bit address, similar to the MAC address of the IEEE 802.xx LAN devices.
AM_ADDR	Active Member Address: Active Member Address - a 3-bit address assigned to active Slaves in a Bluetooth Piconet. The address '0x00' is reserved for broadcast transmissions (point to multi-point) which limits piconet capacity to 7 active Slaves.

PM_ADDR	Parked Member Address: An 8-bit address assigned to Parked Slaves in a Bluetooth Piconet.
BB_PDU	Baseband protocol data unit
LMP_PDU	Link manager protocol data unit
L2CAP_PDU	Logical link & adaptation layer protocol data unit
HCI_PDU	Host controller interface
slot	Slots define 625 microseconds in the time domain and are the basic construct of Bluetooth Frames.
ACL links	Asynchronous Connection-Less Link: ACL links utilize the Bluetooth protocols for data connections. These links are packet switched meaning that the Master can address any given packet to any active Slave.
SCO link	Synchronous Connection Oriented Link: A symmetric, point to point link between a Master and a specific target Slave that is analogous to a circuit switched connection.
Connection Handle	Unique number tagging every active Bluetooth connection.
ISR	Interrupt Service Routine
mC	Microcontroller
RISC	Reduced Instruction Set Computer: a microprocessor that is designed to perform a smaller number of types of computer instructions so that it can operate at a higher speed (perform more millions of instructions per second - MIPS).
ASSP	An Application-Specific Standard Product is a semiconductor device integrated circuit (IC) product that is dedicated to a specific application market and sold to more than one user (and thus, "standard"). The ASSP is marketed to multiple customers just as a general-purpose product is, but to a smaller number of customers since it is for a specific application. Like an ASIC (application-specific integrated circuit), the ASSP is for a special application, but it is sold to any number of companies. (An ASIC is designed and built to order for a specific company.)
IT	Information Technology is a term that encompasses all forms of technology used to create, store, exchange, and use information in its various forms (business data, voice conversations, still images, motion pictures, multimedia presentations, and other forms, including those not yet conceived). It's a convenient term for including both telephony and computer technology in the same word. It is the technology that is driving what has often been called "the information revolution."
I2C	The I2C (Inter-IC) bus is a bi-directional two-wire serial bus that provides a

	communication link between integrated circuits (ICs). Phillips introduced the I2C bus 20 years ago for mass-produced items such as televisions, VCRs, and audio equipment. Today, I2C is the de-facto solution for embedded applications. There are three data transfer speeds for the I2C bus: standard, fast-mode, and high-speed mode. Standard is 100 Kbps. Fast-mode is 400 Kbps, and high-speed mode supports speeds up to 3.4 Mbps. All are backward compatible. The I2C bus supports 7-bit and 10-bit address space devices and devices that operate under different voltages.
WASP	A wireless application service provider (WASP) is part of a growing industry sector resulting from the convergence of two trends: wireless communications and the outsourcing of services. A WASP performs the same service for wireless clients as a regular application service provider (ASP) does for wired clients: it provides Web-based access to applications and services that would otherwise have to be stored locally. The main difference with WASP is that it enables customers to access the service from a variety of wireless devices, such as a smart-phone or personal digital assistant (PDA).

(Note: Most of the above terms have been drawn from the "WhatIs?com" online term encyclopedia and the Xilinx online "Bluetooth Glossary".)