Christoph Wolf

Infrared Communication for a Wearable Computing and Communication System

Master Thesis

Institute for Applied Information Processing
and Communications - IAIK
Technical University Graz, Austria

Advisor:

Prof. Gerald Q. Maguire Jr.
Computer Communication Systems Laboratory
Department of Teleinformatics
KTH Stockholm, Sweden

Examiner:

DI Dr. Karl Christian Posch, IAIK

Graz, November 2000

Abstract

Prof. Gerald Maguire, head of the Computer Communication Systems Laboratory (CCSlab) at the Department of Teleinformatics, KTH, Stockholm, Sweden and Dr. Mark. T. Smith, HP Labs, Palo Alto, California, USA have developed a slightly larger than ID-card sized wearable computer, the so-called SmartBadge. This device is designed for low power consumption and equipped with a 200 MHz RISC processor, 1 MB of FLASH and 1 MB of SRAM memory, various communication channels, audio input and output, video output, and a socket for PCMCIA cards, which allows to use for example wired or wireless network cards. In addition to this equipment, which is comparable to other PDA-like devices it also contains a variety of sensors. These sensors allow it to extend its range of possible applications by adding knowledge about the environment which can be used to provide new context- and location-aware services.

In the first chapters of this thesis I give an introduction to the hardware platform. The operation and configuration of several processor units is explained and their use is shown via a set of example routines and programs.These samples together with their explanations allows a new user to quickly write simple applications.

The second part of this thesis, which comprises the main work of this thesis, concerns a port of the Linux IrDA-stack to the SmartBadge. The IrDA-stack is a set of protocols that allow wireless communication between devices via an infrared link. These protocols are already widely used for data exchange between Laptops (and increasingly also PDAs), mobile phones and other mobile devices. Together with the use of Infrared-Access-Points this implementation provides wireless network-connectivity for the Badge. Through such wireless network access the possible range of applications again can be significantly extended. After an introduction to the relevant protocols I present the details of my implementation. I conclude with a summary of why fitting these various pieces of hardware and software together was difficult and provide some hints of future work.

Acknowledgments

The work for this Master Thesis has been carried out in the Computer Communication Systems Laboratory (CCSlab) at the Department of Teleinformatics, Royal Institute of Technology (KTH) in Stockholm, Sweden during the period of February to July and October 2000.

I would like to express my sincerest gratitude to my advisor Prof. Gerald Q. Maguire Jr., head of the CCSlab, who guided me throughout the course of work for the thesis at hand. I deeply thank Prof. Maguire for his continuous and extraordinary support at work. Moreover, I would like to extend my thanks and appreciation for the many hours of interesting and instructive discussions, which widened my views and ways of thinking in many new aspects.

I would also like to thank my supervisor and examiner from the IAIK Graz, Dipl. Ing. Dr. Karl Christian Posch, for establishing the initial contact with Prof. Maguire.

Further thanks go to my friends and colleagues at the Teleinformatics Department, especially Enrico Pelletta, Iyad Al-Khatib and Pawel Wiatr, for their encouragements, patience, and continuous support by listening to my problems. Special thanks go to Alberto Escudero Pascal for his extensive help with all kinds of Linux related problems.

In addition, I would like to thank KTH's External Relations Office and the people from the ISS, the KTH Student Union Section for international students, for their efforts to make the exchange students' life in Stockholm easy, pleasant, and eventful.

Finally I would like to express my deepest thanks for the support and encouragement I have received from my family throughout my whole studies.

Table of Contents

List of Figures

List of Tables

Listings

1. Introduction

With ever growing computing power and increasing levels of integration which leads to the ever decreasing size and price of electronic components -- mobile applications are increasingly important. PDAs already offer impressive computing power and mobile phones are a fixture of everyday's life. Increasingly these advances are leading to a fusion of PDA like devices with communication. This is compounded with the explosive growth of the internet.

Prof. Gerald Maguire, head of the Computer Communication Systems Laboratory (CCSlab) at the Department of Teleinformatics, KTH, Stockholm, Sweden and Dr. Mark. T. Smith, HP Labs, Palo Alto, California, USA have developed a wearable computing platform that was optimized in terms of power consumption and size, but yet offers state of the art computing power. In addition to that they have integrated a number of sensors for measuring light, temperature, humidity, and acceleration. This allows one to investigate and develop a new class of applications and services. The new concept is that the applications now can acquire information about their environment. This results in so called context- and location-aware services. Some location aware services have already been introduced, for example mobile systems that detect their location (e.g. by means of the global positioning system (GPS) or by other means provided by the cellular network infrastructure) and based upon the knowledge of their position can offer information services that provide (only) information that is relevant at a certain location (this could be information about shops, restaurants, public services, localized traffic information and many more).

The sensors also add an additional dimension. They can detect if the device is still attached to its user (which could be used for authentication mechanisms). As long as the users are still wearing their device they do not need to reauthenticate. Other applications are e.g. medical monitoring. Combining these additional technologies and a communication infrastructure allows services that are based on both the location and the context of a user.

In addition these so called SmartBadges are equipped with audio input and output, an IR transceiver, and can utilize PCMCIA-form factor network adapters (the later two can provide wireless access to network infrastructure). All this put together results in a huge field of possible applications.

The SmartBadge as used in this thesis was version 3, a new version 4 is to be introduced soon. In the course of this thesis work my main field of work involved enabling infrared communication, with the final goal to provide network access via an infrared link.

After a short introduction to the hardware platform and the available operating systems I present the use of some of the processor units such as: serial communication, general purpose Input/Output, timers, etc. Later chapters describe some aspects of the code generation of the compiler and the problems of debugging embedded systems. The final chapters present the main result of my work, an implementation of the IrDA protocol stack which in combination with an infrared access point (that is attached to a fixed network) provides the SmartBadge with network access via an infrared link.

2. Basic Architectures

In this chapter I give an introduction to the hardware architectures of both SmartBadge 3 and the new SmartBadge 4. As most of my work was based on SmartBadge 3, I also give a short overview of the StrongARM SA-1100 microcontroller which is used on Badge 3. Its successor, the SA-1110, used on Badge 4, differs from the SA-1100 only in the memory interface (which now also supports SDRAM and allows SRAM, DRAM, and SDRAM in the same system) and the PCMCIA controller. The SA-1111 companion chip provides the PCMCIA controller as well as support for Compact Flash, a complete USB Host Controller, two serial ports (Serial Audio Controller and SSP Serial Port), PS/2 Trackpad and Mouse Interfaces and an additional general purpose I/O Interface.

2.1 SmartBadge 3

SmartBadge 3 [1] consists of the following basic blocks as shown in Figure 1:

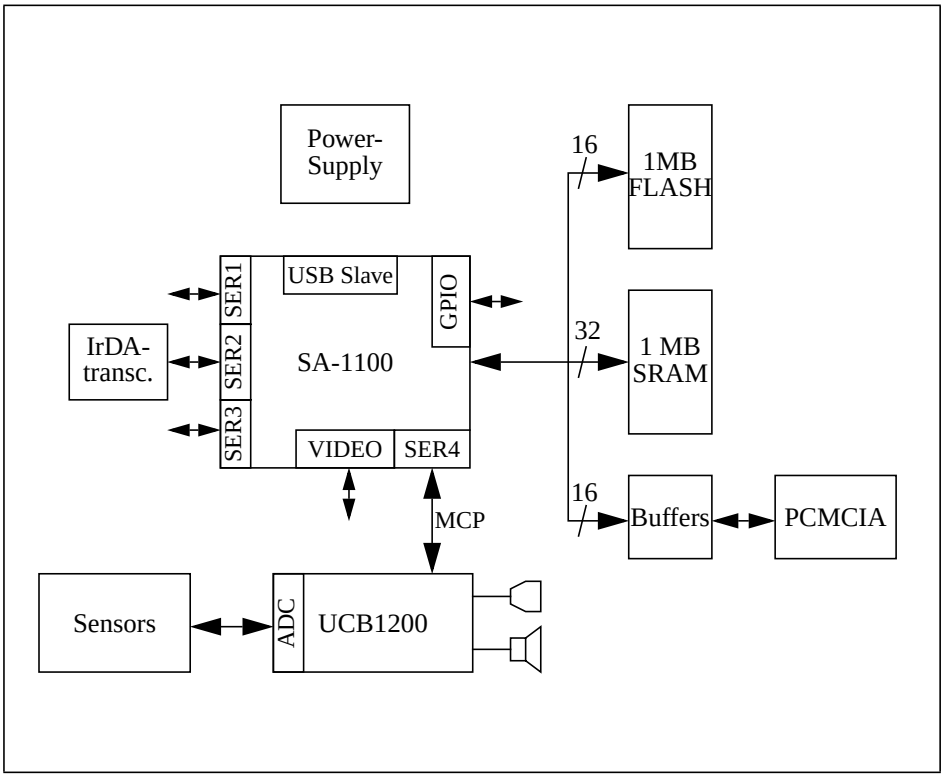


Figure 1. SmartBadge 3 Block Diagram

The central element is an Intel StrongARM SA-1100, a 32 bit RISC controller clocked at up to 200MHz. It is connected to 1 MB of FLASH memory (16 bit), 1 MB of SRAM (32 bit) and an audio and telecom codec (Philips UCB 1200). The SA-1100 is described in more detail in the next section.

The circuit can be powered by onboard batteries or - for test and development - by an external power supply. The power supply block takes this unstabilized voltage and generates 1.5V for the microcontroller core, 3.3 V for the microcontroller peripherals and the other circuits and optionally 5V for the PCMCIA slot if a 5V card is used. The buffers between the SA-1100 and the PCMCIA connector are necessary to convert between the 3.3 V system supply and the 5V PCMCIA supply, when a 5V card is being used.

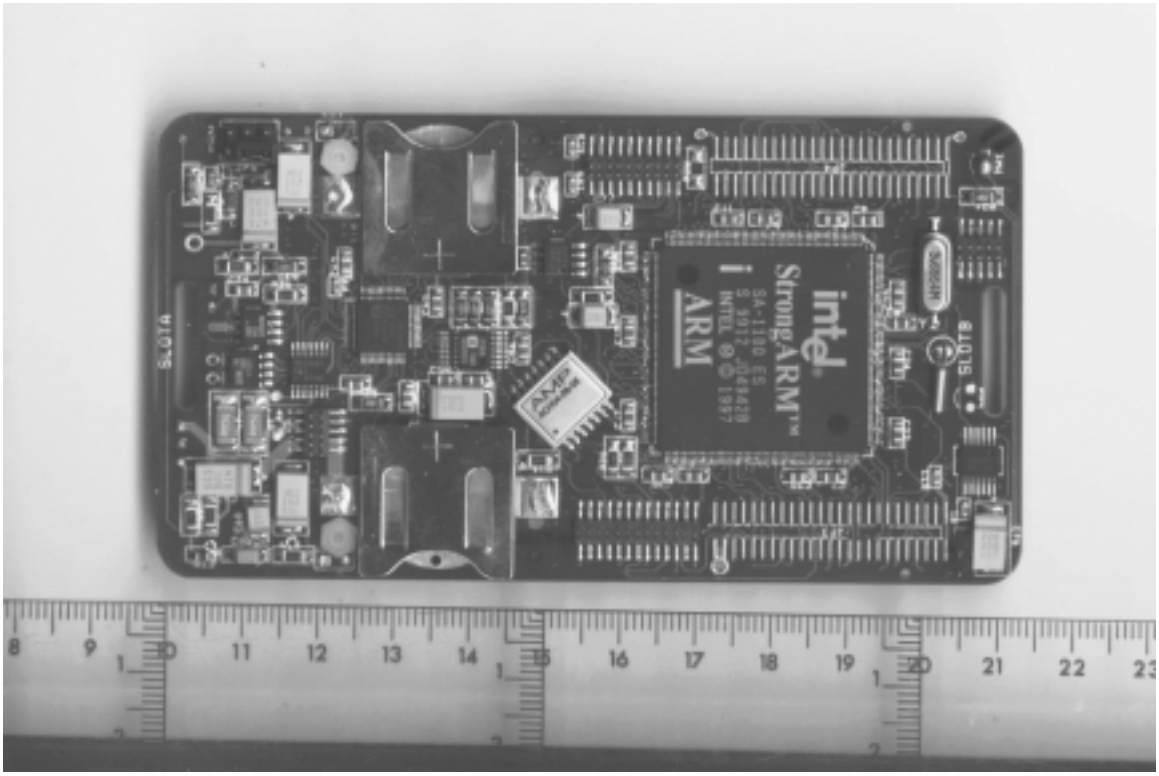
The FLASH memory is 16 bit wide while the SRAM memory utilizes the full width of 32 bit. Although the SA-1100 also supports DRAM, SmartBadge 3 doesn't provide any DRAM, mainly to reduce the power consumption.

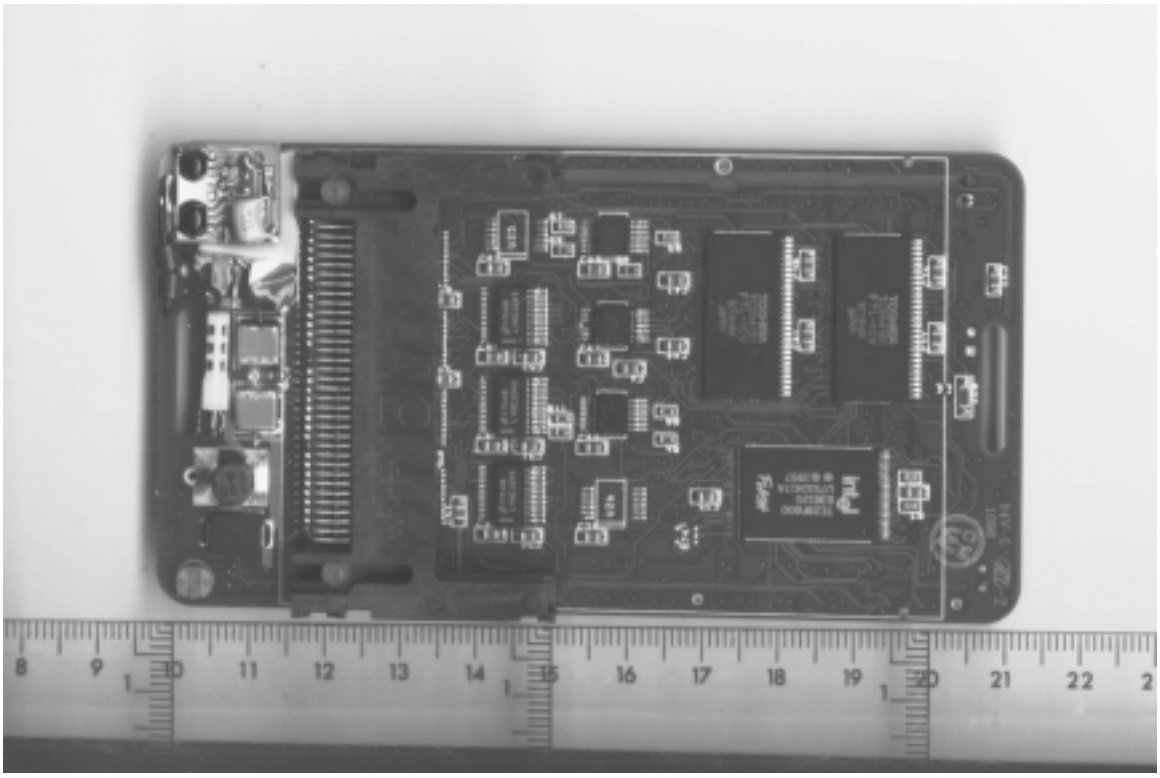
The Philips UCB1200 [2] is a single chip, integrated mixed signal audio and telecom codec. It allows direct connection of a microphone and a speaker via the audio codec channel. The telecom codec channel can be directly connected to a DAA and supports high speed modem protocols. These two channels mainly differ in the output impedance, therefore if the telecom functionality is not required the two channels can be combined to form stereo audio input/output. In addition the UCB contains a built-in 10-bit ADC. A built-in analog multiplexer allows selection between four general purpose analog inputs and a four input touch screen interface. Finally there are ten digital general purpose I/O-pins available. On SmartBadge 3 the general purpose ADC-inputs and two of the touch screen interface inputs are used to read in the analog sensor values while the digital I/O-pins allow selectively powering the sensors. The UCB1200 is connected to the SA-1100 via the multimedia serial port (serial port 4, MSP mode).

The following sensors are included by default, additional sensors can be connected via the general purpose I/O-pins or the serial ports, if required:

- three-axis accelerometer,
- two humidity sensors,
- two temperature sensors,
- a light sensor.

The light sensor is mounted on the front side, as well as one of the humidity sensors and one of the temperature sensors. The other sensors are mounted on the backside. This allows the system to differentiate between sensors on the side orientated towards the user's body and the side orientated away from the body if the Badge is worn, using one of the two slots in the Badge - as shown in the two photos below:





2.2 Description of the StrongARM SA-1100 Microcontroller

The StrongARM SA-1100 [6] is a highly integrated high-performance low-power microcontroller, especially suited for (mobile) telecommunication applications. It consists of a 32-bit StrongARM RISC processor core, extended by system support logic (memory-, DMA-, interrupt-, power-management-controllers, timers and real-time-clock), caches, various communication-channels, an LCD controller, PCMCIA controller, and general purpose input/output ports. Due to its special low-power design it has a typical power dissipation of about 200 mW @ 200 MHz. For further power saving the clock can be slowed or stopped under software control. Logically these elements can be grouped into the following main modules:

- Processor core
- MMU, caches, read and write buffer
- System Control Module (General Purpose I/O, Interrupt Controller, Real Time Clock, Operating System Timer, Power Manager, Reset Controller)
- Memory and PCMCIA Control Module
- Peripheral Control Module (DMA Controller, LCD Controller, Serial Ports 0-4, Peripheral Pin Controller)

Figure 2 gives an overview of these basic components:

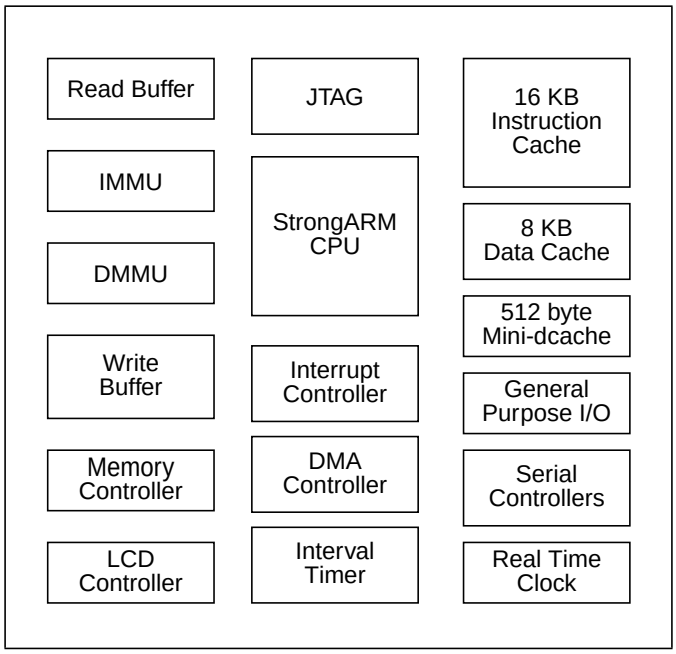


Figure 2. Components of the StrongARM SA-1100

In the following sections I will give a short overview of the caches, buffers and MMU. More detailed information can be found in [5] and [6]. Some of the peripheral units will be described in more detail in a later chapter (See 4. “Using some of the Peripherals“).

2.2.1 Instruction Cache

The instruction cache (IC) has a size of 16 kBytes, set up as 512 lines of 32 bytes (8 words) and is arranged as a 32 way set associative cache. It can be enabled or disabled via the SA-1100 Control Register and is disabled through a reset sequence (i.e. on the assertion of the nRESET line, software or sleep). It’s operation further depends on the state of the Memory Management Unit and the Cacheable bit stored in the Memory Management Page Table. The cache works as follows:

The IC operates with virtual addresses, therefore care must be taken to ensure consistency with the MMU mappings, special care is needed if the mappings are changed. The IC is not coherent with stores to memory. If a program writes to cacheable instruction locations not only must the cache operation be taken into consideration but also the write buffer (see section 2.2.3) must be drained as instruction fetches do not check the write buffer.

If the cache is disabled, no lines are placed in the cache, but the cache is always searched and data found will be used by the processor - only cache misses are affected by the state of the cache, the MMU and the Cacheable (C) bit. This is particularly important if the cache is disabled by software after having been enabled for some time. If the data in the cache must not be used the cache must be flushed.

In the case of a cache miss the behaviour depends on the state of the MMU and the Cacheable bit:

- MMU disabled or C bit is set for the given virtual address: a linefetch of eight words is performed and placed in a cache bank with a round-robin replacement algorithm.
- MMU enabled and C bit is zero for the given virtual address: an external memory access for a single word is performed and the cache is not written.

Note: if Memory Management is disabled, all addresses are regarded as cacheable (i.e. C=1).

2.2.2 Data Caches

The SA-1100 contains two logically separate data caches:

- main data cache: intended for use during most data accesses
- mini data cache (also called mini-cache): alternate caching structure for dealing with large data structures which could thrash the main cache.

Both caches use virtual addresses and allocate only on loads, write misses never allocate in the cache. Besides the actual data each cache line contains also the physical address of this line and two dirty bits. The dirty bits indicate the status of the first and the second half of the line. Store hits in the cache cause the associated dirty bit to be set. When a line is evicted from the cache the dirty bits are used to decide which part of the line (all, half, or none) has to be written back to memory using the physical address stored together with the line. Both caches always reload a complete line (8 words) at a time.

As for the instruction cache the data caches can be enabled and disabled via the SA-1100 Control Register and are disabled by resets (including watchdog reset). Apart from this global enabling/disabling the operation depends on the Bufferable (B) bit and the Cacheable (C) bit stored in the Memory Management Page Table. Cache operation therefore requires the MMU to be enabled. Again as virtual addresses are used by the caches consistency in the virtual to physical mappings performed by the MMU must be ensured. In particular it is assumed that every virtual address maps to a different physical address. Doubly-mapped virtual addresses should be marked as uncacheable to avoid cache inconsistencies (each virtual address has a separate entry in the cache and only one entry is updated on a write operation).

Typically main memory is marked as cacheable whereas I/O space should always be marked as uncacheable to make sure that the hardware registers are always directly read instead of copies of earlier values stored in the cache.

2.2.2.1 Main Data Cache

The main data cache is an 8 kByte writeback data cache. It consists of 256 lines of 32 bytes (8 words) in a 32 way set associative organization (i.e. 8 sets, each consisting of 32 blocks of 8 words) . It allocates on loads to memory locations marked as B=1 and C=1. Replacements in the main data cache are selected according to a set of round robin pointers. At reset the pointers in each set of the cache point to block zero of each 32-block set. As lines are allocated, the pointers are incremented to the next block. After block 31 has been allocated, the next line fill replaces (and copies back to memory, if dirty) the data in block zero.

2.2.2.2 Mini Data Cache

The mini-cache is a 512 byte writeback cache consisting of 16 lines of 32 bytes (8 words) in a 2 way set associative organization. It allocates on loads to memory locations marked as B=0 and C=1. Replacements in the mini data cache also use a round robin pointer mechanism. But, since this cache is only two way set associative, the replacement algorithm reduces to a simple Least-Recently-Used (LRU) mechanism.

2.2.2.3 Detailed Operation with Respect to the C and B Bits

Cache hits are always served, i.e. on a load cache hit the according cache delivers the data and on a store cache hit the data is stored to the according cache and the line is marked as dirty.

As the caches only allocate on load misses, on a store miss the data is stored to memory without affecting the cache (no allocation).

In the case of a load cache miss the operation depends on the C and B bits as follows:

- C=0: load from memory, no cache allocation
- C=1 and B=0: load from memory and allocate to mini cache
- C=1 and B=1: load from memory and allocate to main cache

		load		store	
B	**C**	**cache hit**	**cache miss**	**cache hit**	**cache miss**
0	0	deliver cache data	load from memory - no allocate	store to either cache - mark line dirty	store to memory, no allocate
0	1	deliver cache data	allocate to mini cache	store to either cache - mark line dirty	store to memory, no allocate
1	0	deliver cache data	load from memory - no allocate	store to either cache - mark line dirty	store to memory, no allocate
1	1	deliver cache data	allocate to main cache	store to either cache - mark line dirty	store to memory, no allocate

Table 1. Data cache operation

2.2.2.4 Data Cache Flush

The SA-1100 supports flush and clean operations on single entries of the data caches as well as flushing the whole cache by writes to the Cache Operations registers. But as the caches are writeback caches, in order to prevent the loss of data, a flush whole must be preceded by a sequence of loads to cause the cache to write back any dirty entries. The memory controller in the SA-1100 provides an internally decoded memory space (residing in the upper 512 Megabytes of the memory map, starting at virtual address 0xE000 0000) that returns zeros without incurring external memory latency.

2.2.3 Write Buffer

The SA-1100 contains a write buffer to improve system performance by buffering up to eight blocks of data of 1 to 16 bytes at independent addresses. The buffer can be globally enabled or disabled via the SA-1100 Control Register. Its operation further depends on the Cacheable and Bufferable bits in the Memory Management Page Tables, therefore the MMU must be enabled in order to use the write buffer.

In detail it operates as follows:

When the CPU performs a store, first the data caches are checked. If the store hits in one of the caches the write completes in the cache, provided that the protection for the location and the mode of the store allow the write to the cache. The write buffer is not used.

If a store misses in both data caches the action depends on the B bit:

- B=1 (and write buffer enabled): The data is placed in the write buffer and the CPU continues execution. The write buffer performs the external write some time later.
- B=0 (or write buffer disabled): On write to an unbufferable area the processor is stalled until the write buffer empties and then the write completes externally. This requires several clock cycles.

2.2.4 Read Buffer

The SA-1100 contains a software programmable read buffer which can increase the performance of critical loops by prefetching data. The read buffer enables the preallocation of read-only data into one of four 32 byte buffers without stalling the pipe. For subsequent loads which hit in the read buffer, data is sourced from the buffer instead of the data cache at a rate of 1 word per core clock. As the data to be contained in the read buffer is explicitly specified in software, critical data can be locked in.

2.2.5 Memory Management Unit (MMU)

The SA-1100 implements the standard ARM memory management functions using two 32 entry fully associative Translation Buffers (TBs). One is used for instruction accesses and the other for data accesses. On a TB miss the translation table hardware is invoked to retrieve the translation and access permission information from the Memory Management Page Tables. Once retrieved, if the entry maps to a valid Page or Section then the information is placed into the TB. The replacement algorithm in the TB is round robin. For an invalid page or section an abort is generated and the entry is not placed in the TB. More detailed information can be found in [5].

2.3 Peripheral Devices in the StrongARM Architecture

While the memory and PCMCIA controller, DMA controller and the LCD controller are directly attached to the ARM System Bus, the peripherals, such as e.g. serial ports, timers, general purpose IO, are attached to the ARM Peripheral Bus, which is coupled to the ARM System Bus via a bridge (see Figure 3). All the on-chip devices and controllers are accessed using memory mapped I/O, that is, the control and data registers are mapped into memory and are accessed using the memory load/store instructions.

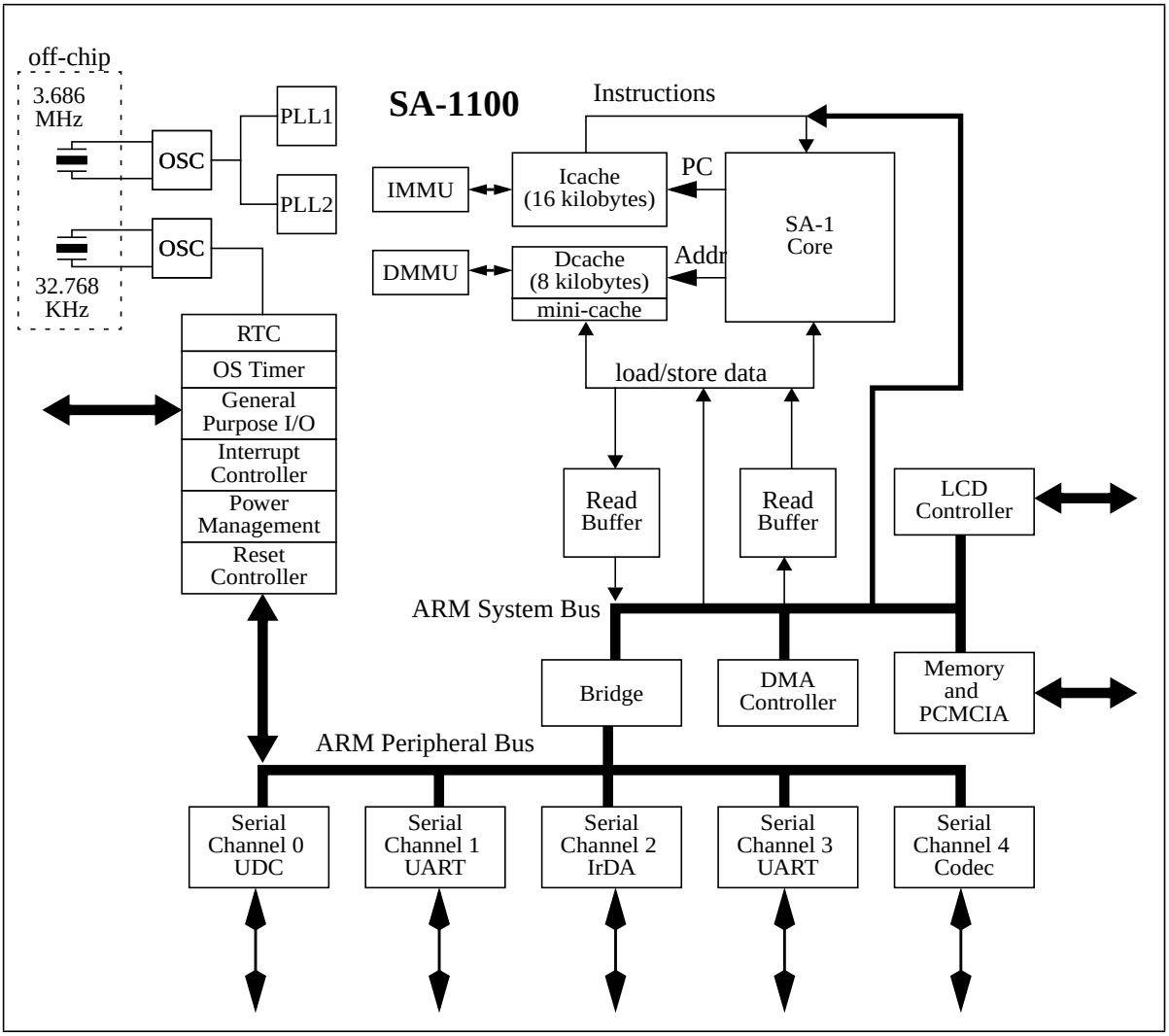


Figure 3. SA-1100 Block Diagram (Note that the two crystals are off-chip)

Figure 4 gives an overview of the SA-1100 memory map:

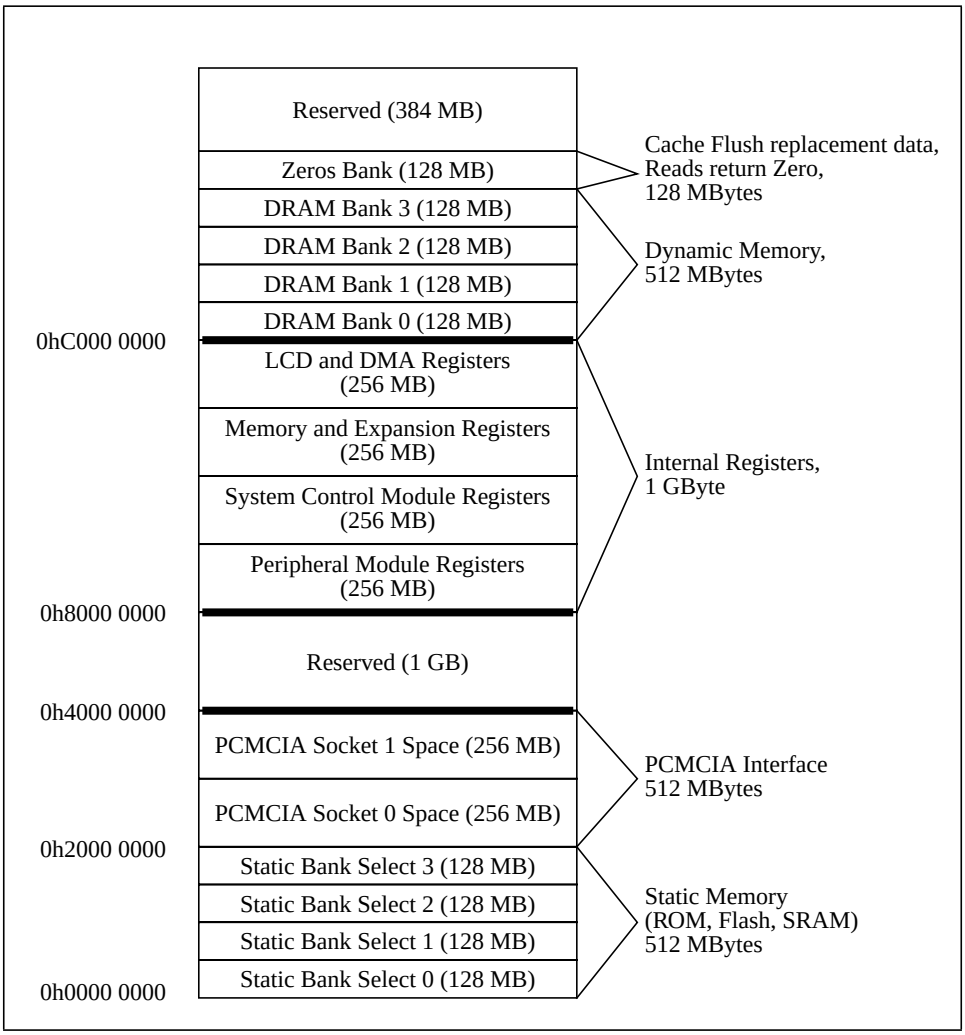


Figure 4. SA-1100 Memory Map

2.4 SmartBadge 4

SmartBadge 4 uses the SA-1110 microcontroller [7] and its companion chip, the SA1111 [8]. The basic building blocks are shown in Figure 5:

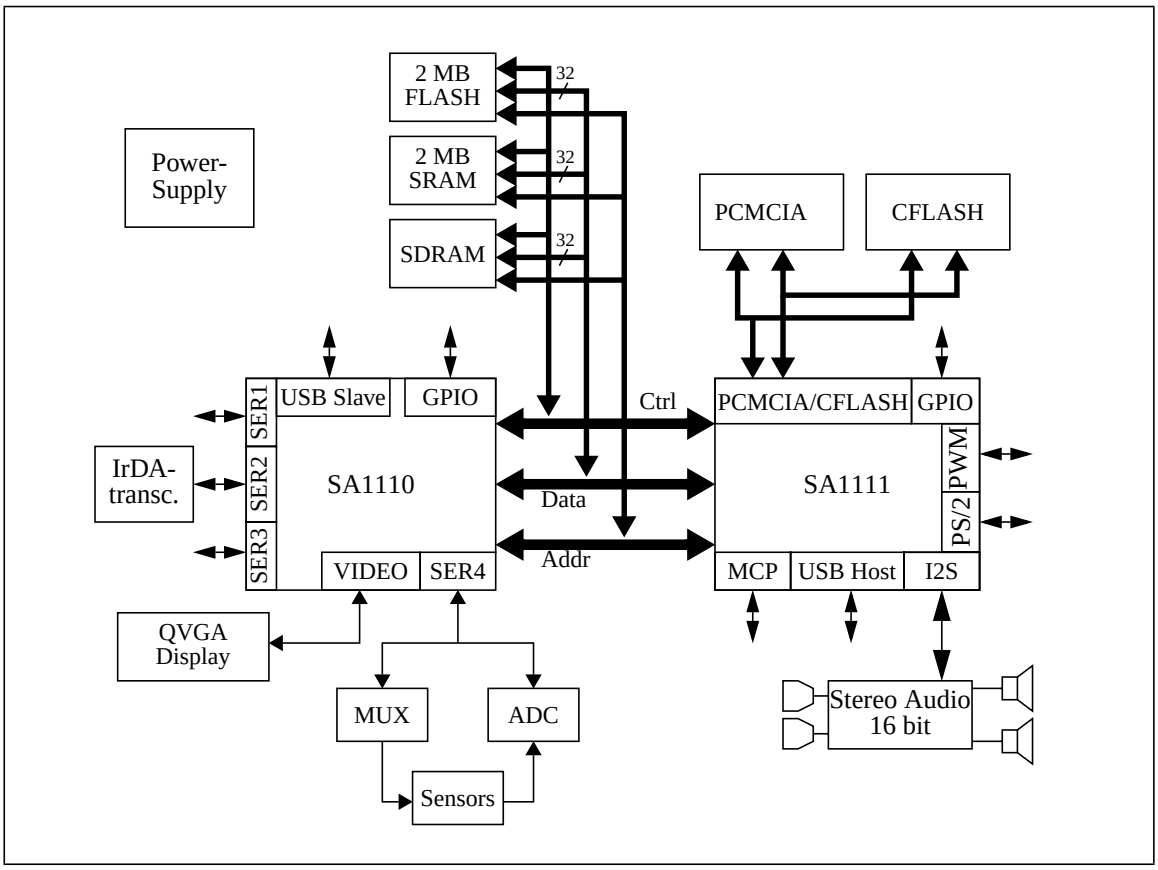


Figure 5. SmartBadge 4 Block Diagram

Version 4 offers the following improvements and extensions over SmartBadge 3:

- Flash memory and SRAM both doubled to 2 MBytes each, and the data path from the FLASH is now 32 bits wide
- Support for (optional) additional SDRAM memory
- CFLASH slot in addition to PCMCIA slot
- Improved audio support now offers 16-bit 44kHz stereo input/output and better shielding
- Port for stereo heads-up display
- Dedicated ADC, separated from digital circuitry to achieve better noise immunity
- USB master, optional BlueTooth interface via USB

3. Operating Systems for the Badge

In this chapter I will briefly describe the three operating systems that currently support the SmartBadge. Personally I worked only with Angel, while VxWorks and E-Kernel were used by two groups of students during the Telecommunications "Fingercourse" at KTH in spring 2000. Thus I will describe Angel in more detail while I will give only a short summary of the experiences with the two other operating systems. HP is also working on porting Linux to Badge 4.

3.1 Angel

3.1.1 Introduction

Angel is a software packet specifically designed to aid the development of ARM based appliances. An Angel system typically consists of two main components:

- Debugger: The debugger is executed on a host computer which is connected to the target hardware via a communications link, typically a serial link but IP/UDP over ethernet is also supported. A Windows-based debugger is included in the ARM Software Development Toolkit, but any other debugger capable of handling the "Angel Debug Protocol" (documented in [3]) can be used. The debugger controls downloading images to the target, executing programs and reading/setting memory locations on the target platform.
- Angel Debug Monitor: This is software that runs on the target and communicates with the host debugger. The debug monitor can be built in two versions:
  - a full version, including support for the debugger, semihosted operations (See 3.1.2), basic operating system functionality; intended for use on development hardware.
  - a minimal version for use on production hardware.

The two versions of Angel allow a smooth migration from development to production hardware. The typical development cycle for ARM based software using the ARM Tools, as described in chapter 6.2 of [4], then is as follows:

- **Evaluation of an application using the ARMulator**, a cycle-exact emulator for ARM processors. As the ARMulator only emulates the ARM-core (but not for instance the additional devices included in the StrongARM) this only works if no interrupts are used and no external hardware is accessed.

or

- **Evaluation of an application on a PIE board under full Angel**. The PIE board is an ARM-based development board running a full version of Angel. It's not necessary to customize Angel, applications can be downloaded, executed and tested using an ARM debugger.

- **Building applications on a custom development board, highly dependent on Angel**. To run on a custom board the low level layers of Angel have to be ported to support the actual hardware. Then applications can be built and tested with the support of the full Angel version, i.e. using the semihosted operations, Angel device driver framework, and others. The SmartBadge is an example for such a custom board running full Angel.
- **Building applications on a custom development board, little dependence on Angel**. After the application has been developed and evaluated using full Angel it can be changed to little dependence on Angel. The board is still running full Angel to support the use of the debugger but the application itself hardly uses Angel features any more.

- **Moving the application to production hardware**. Now that the application doesn't rely on Angel any more Angel can be rebuilt as minimal version. Minimal Angel is structured the same way as full Angel, i.e. initialization, device drivers, interrupt support work the same, but features such as debugging, semihosting, multiple channels on one device are no longer supported. This frees up resources (memory need, communication links,...) which are not needed any more and are scarce on production hardware.

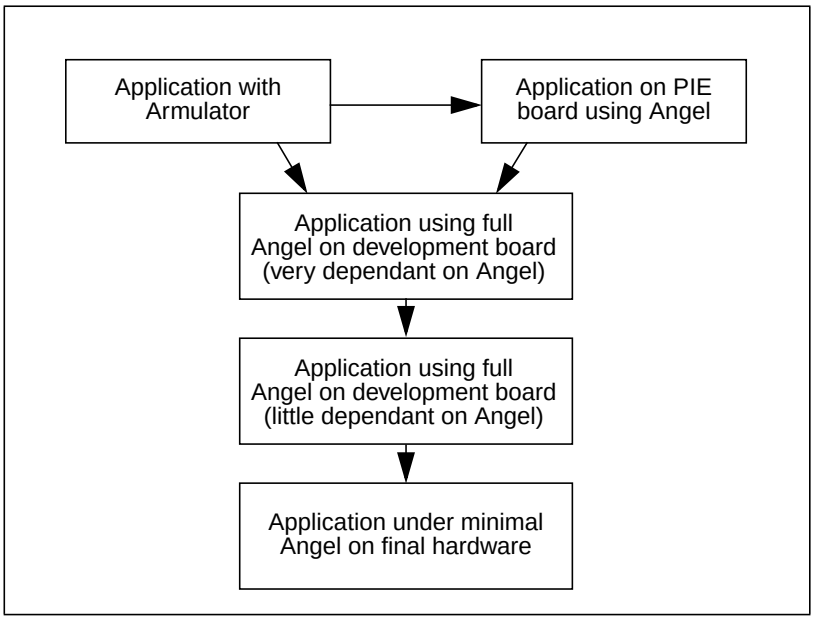


Figure 6. The Angel development cycle [4, p. 6-4]

3.1.2 Angel C Library Support, Semihosted Operations

The C Library for Angel is split into two parts, the C Library itself which is linked with the application, and support for the semihosted parts of the library - this part is linked with Angel.

Angel doesn't provide a symbol file to link applications against, therefore Angel functions cannot be called directly from user applications that have been downloaded to the target via the debugger. Through the use of software interrupts (SWI) Angel allows such applications to make requests. Some of these requests are semihosted, i.e. a request from the user program is communicated to the host and executed there.

To avoid interfering with operating systems that also use SWIs Angel uses only one SWI for all operations. The particular service that is requested is encoded in register r0, the parameters are passed in a block which is pointed to by register r1. The result is returned in r0, either as a value or as a pointer to a data block.

The supported semihosted operations are as follows [3, p. 8-3 ff.]:

- SYS_OPEN: open a file on the host computer
- SYS_CLOSE: close a previously opened file
- SYS_WRITEC: write a byte to the debug channel, when executed under debugger control the byte appears on the display device connected to the debugger
- SYS_WRITE0: write a 0-terminated string to the debug channel, appears at the debugger display device
- SYS_WRITE: write a data block to a previously opened file on the host
- SYS_READC: read a byte from the debug channel, i.e. from the keyboard attached to the debugger
- SYS_READ: read a block of data from a previously opened file on the host
- SYS_ISERROR: check a status word

- SYS_ISTTY: check if a handle to a previously opened file or device object identifies an interactive device
- SYS_SEEK: set the position in a seekable file
- SYS_FLEN: return the length of a seekable file
- SYS_TMPNAM: get a temporary filename from the host
- SYS_REMOVE: delete a file on the host
- SYS_RENAME: rename a file on the host
- SYS_CLOCK: return the time since the support code started executing, queried from the host
- SYS_TIME: return the number of seconds since the start of 1970
- SYS_SYSTEM: pass a string supplied in a buffer to the host's command interpreter and return the status
- SYS_ERRNO: return the value of the C library variable `errno` associated with the host support
- SYS_GET_CMDLINE: return a string of the command line used to call the executable

Other operations intended to be used by user applications:

- SYS_HEAP_INFO: return info about stack and heap base and limit
- angel_SWIreason_EnterSVC: on return the processor will execute in SVC (supervisor) mode with interrupts disabled. It returns the address of a function to be called to return to USR (user) mode (Angel_ExitToUSR).
- angel_SWIreason_LateStartup: Angel supports late startup for the debugger, i.e. the application starts standalone and upon need can request the debugger to be started to inspect an error condition.
- angel_SWIreason_ReportException: allows the application to report an exception directly to the debugger, e.g. that execution has completed.

3.1.3 Communications Architecture for Angel

Figure 7 shows a model of the communication layers for Angel.

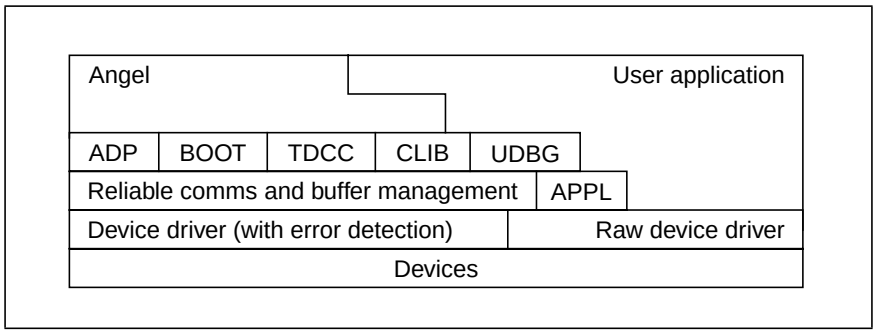


Figure 7. Communication layers for Angel [3, p. 8-10]

At the lowest level Angel uses Devices. Normally there is at least a device using a serial link but other devices are also possible. For example, for a certain development board (ebsa110) which contains an ethernet controller, Angel can be configured to include the Fusion IP/UDP stack, which then allows debugging via ethernet. To support this, low level device drivers for the used ethernet controller are provided, which the IP/UDP stack uses to transmit/receive. To make the ethernet accessible for Angel an additional wrapper layer was added which models an Angel device using the provided socket functions to access the IP/UDP stack. From an Angel point of view the whole network stack is just another device which can be accessed through the common Angel device framework. The device driver layer provides detection or rejection of bad packets but doesn't offer reliability.

The device driver multiplexes reliable packets from Angel with raw packet access from the application. This makes the transition from full Angel to minimal Angel easier - the interface for the application (raw device) remains the same, the raw device driver then just directly accesses the link.

All communications for debugging (the channels ADP (Angel debug protocol), BOOT (boot agent channel), TDCC (Thumb direct comms channel), CLIB (semihosted C library support), UDBG (user debug support for extended debugger features accessible for the application)) require a reliable channel between the target and the host. The “Reliable comms and buffer management” layer implements reliability, retransmission and multiplexing/ demultiplexing for these channels. To allow retransmissions after errors this layer also implements buffer management. Reliability is achieved by using sequence numbers in the packets that allow to detect missing packets and retransmission of these or corrupted packets.

At the top level, Angel communicates with the debugger through the described channels, the user application can request semihosted operations (CLIB) or extended debugger features (UDBG) and can use the device connected to the debugger host for its own communication (APPL) and/or other devices.

Angel supports polled devices, interrupt driven devices and half interrupt driven/half polled devices (i.e. the operation is started via an interrupt but then completed by polling). As packet processing can result in time consuming operations, this can cause problems if executed within interrupt handlers. To avoid blocking the system by performing lengthy tasks within an interrupt handler Angel provides a serialization mechanism. As this mechanism proofed vital for my IrDA implementation I’ll describe it in some more detail in the next section.

3.1.4 Serialization and CPU Modes

The serialization model used by Angel is explained using an example:

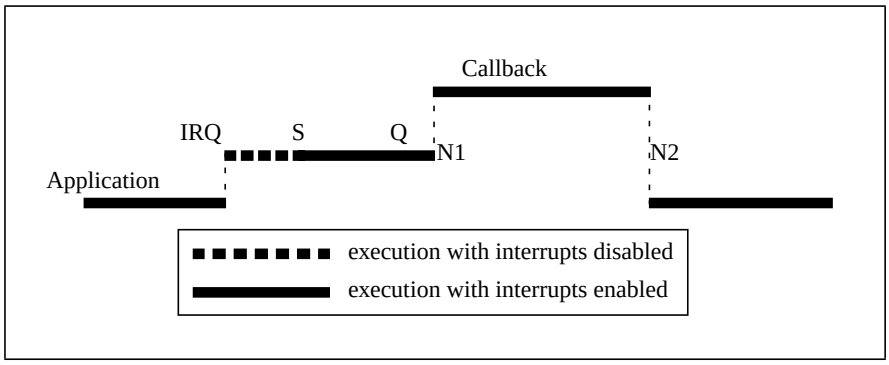


Figure 8. Serialization [3, p. 8-38]

The application is running, when an interrupt request (IRQ) occurs. Angel stores the application stack and task info, disables interrupts, switches the CPU mode to IRQ and executes the interrupt handler. The interrupt handler calls `Angel_SerialiseTask` (S) and provides a function as an argument. Angel keeps a structure called “the lock” to achieve mutual exclusion. If the lock is currently “unowned”, the passed function is immediately executed in SVC (supervisor) mode, but with interrupts reenabled. If the lock has already been taken by another call, the complete context information to execute the function later on, when the lock has been returned, is saved. From point “S” on the function executes in SVC mode. The rest of the packet then can be polled in and the appropriate callback function can be determined. Now `Angel_QueueCallback` is called (Q) to queue a callback function for to process the packet for later execution. When the packet is complete the packet assembly code (i.e. the serialized function) returns. This is intercepted by `Angel_NextTask` (N1). Instead of returning to the application the queued callback function is executed in USR mode. When this has finished it returns. Again the return is intercepted by `Angel_NextTask` which returns execution to the application.

Through this mechanism interrupt handlers can postpone execution of potentially time consuming code to a later time and thus minimize the time spent with interrupts disabled. The serialization provides mutual exclusion because

only one function at a time can grab the lock. On each return of a task Angel checks the callback queues and executes the queued requests according to their priority specified in the call to `Angel_QueueCallback`.

3.1.5 Summary

Angel is a debug monitor that allows to download user applications to a target platform via a debugger. The user application can request services on the host running the debugger through the semihosted C library.

Angel provides a device driver framework that allows to add and integrate new devices relatively easily. Using this framework it is possible to smoothly migrate from development hardware using full Angel support to production hardware with minimal Angel support.

The serialization mechanism allows mutual exclusion and to delay time consuming tasks by queuing callback functions for later execution, thus minimizing the time spent with interrupts disabled.

Porting the low level layers of Angel is not an easy task, especially as at this level a logic analyzer is nearly the only possible method of debugging. At higher levels the code is documented quite well, allowing modifications to fit special needs.

A disadvantage is that by default interrupt handlers have to be (statically) compiled into Angel, changing handlers and calling Angel functions from programs downloaded via the debugger is not possible. As explained in section 4.1 “Extensions to Angel” this can be changed by applying some modifications to Angel.

More information on Angel can be found in [3] and [4].

3.2 VxWorks

VxWorks is a commercial real-time operation system by Windriver Software. It has a rich feature list and offers nearly everything one can think of with respect to an embedded operating system, including a full TCP/IP stack. The cost is an enormous image size and high complexity. The student group working with VxWorks spent most of their time trying to configure the OS to make it work instead of being able to concentrate on their own code. The OS was delivered in a new version which the students were not able to configure properly, even with the help of an VxWorks-experienced HP developer. Finally they switched back to the old version which was known to run. Although listed in the feature list it was not possible to configure the OS such that self developed programs could be downloaded via the serial link. They had to be compiled and linked directly with VxWorks, thus requiring rewriting the whole FLASH memory for every change in the program code - a time consuming process. To sum up, VxWorks offers a lot of features but is extremely complex and big in size - typical image sizes including the OS and a small user program were about 800-900 KB, i.e. the OS itself took up most of the available FLASH memory. An example for a working application is a web server running in a SmartBadge equipped with a wireless LAN card, allowing to query the sensor values.

3.3 E-Kernel

E-Kernel is a component OS being developed by Tim Connors, HP Labs, Palo Alto. It consists of various relatively independent modules that can be linked in according to the specific requirements of a given application. At the moment it offers multithreading and modules for console I/O and mutex. In combination with an RPC-based daemon a simple command shell allows to load and start programs stored on the host running the daemon. RPC-daemon also provides semihosted services similar to Angel, i.e. I/O functions like `printf` and `getline` are redirected to the host terminal. HP has also licensed a TCP/IP stack which, in combination with a WaveLAN driver also written by Tim Connors, allows full network access in a wireless infrastructure. The experiences in general were quite good, the OS already offers the basic functionality one can expect. A big advantage over VxWorks is that the size of the OS is extremely small, with the TCP/IP stack included it takes up about 140 KB. Some problems arose due to the still very limited documentation, a result of the early development stage of the OS. A drawback compared to Angel and VxWorks is that, at least until now, there is no debugger hook. With better documentation and especially adding debugger support I’m convinced that the E-kernel will become a valuable alternative.

4. Using some of the Peripherals

This chapter first describes the changes applied to Angel to offer enhanced functionality and then describes some of the peripheral components, such as interrupts, general purpose I/O, peripheral pin controller, timers, UARTs, and infrared port, in more detail. For each unit discussed here first a brief explanation of the controlling registers is given, followed by code examples that demonstrate how to use the unit. Some of the units that the SA-1100 also offers, such as USB device controller, SDLC mode for serial port 1, serial port 4 (multimedia port), LCD controller, power management and reset controller are omitted here, more information on them can be found in [6].

4.1 Extensions to Angel

4.1.1 Description of the Changes Applied to Angel

First I describe an extension to Angel that has been made by Prof. G.Maguire and M.T.Smith, then some more extensions which I have added during my work.

In its current design Angel keeps a table containing the address of a handler function and an additional 32-bit word to be passed to that handler for each of the 32 first level interrupt sources (see section 4.3 “Interrupts”). This table is kept in SRAM but is statically compiled into Angel. Thus no runtime modification by application programs downloaded via the Debug Monitor is possible. To get around this problem, Prof. G.Maguire and M.T.Smith have changed `_syscall 0x16`, which by default returns heap information, to return additional information including the base address of this interrupt handler table - thus making it possible to install/remove interrupt handlers at runtime. The table is located in the file `\Arm211\Angel\Source\brutus\devices.c`, which also contains the added array `sysInfo`.

The mechanism to access the table works as follows: Angel provides no symbol file to link against at runtime. Therefore applications can’t call any Angel functions directly. In order to offer services Angel uses a software trap: executing a SWI (Software Interrupt) instruction causes a trap which is intercepted by Angel. The SWI trap handler calls `SysLibraryHandler` (in file `\Arm211\Angel\Source\brutus\sys.c`) which evaluates some of the processor registers that have been set up suitably in advance. Their contents are interpreted as an index to select a function to call and as parameters to pass to that function respectively. These functions implement the so-called syscalls and carry out the requested service. The services mainly represent the semihosted operations described in section 3.1. One of these functions, `_syscall 0x16`, returns information about the heap. The integer pointer array `sysInfo` was added to the file `devices.c` and contains the address of the Angel device table, the device status table, the interrupt and the polling handler table, and the default interrupt handler. This array was added to the existing variable of type `struct AngelHeapStackDesc` in the file `\Arm211\Angel\Source\stacks.c`. This variable in turn is returned when `_syscall 0x16` is executed. Now that we have the base address of the interrupt handler table it is possible to replace any interrupt handler during runtime.

I used the same mechanism to add some more information to be returned by the above system call. During my work on the IrDA implementation I faced severe problems regarding the CPU mode in which code is executed. Basically the whole IrDA implementation is event driven, either communication events or timer events. The timer events can only be generated using (OS Timer) interrupts and trigger the execution of partly quite time consuming functions. The problem now was that doing the timer check and executing all following actions within the timer interrupt handler would block all other interrupts during this time which is clearly unacceptable. As explained in section 3.1, Angel offers a mechanism which is intended to exactly solve this type of problems by queuing time consuming tasks for later execution in user mode with interrupts reenabled. But again the problem was that these functions could only be used by code that was statically linked with Angel, not by application programs (running under the Debug Monitor). Therefore I extended the `sysInfo` array to contain pointers to the functions I needed. By executing `_syscall 0x16` these function pointers are returned and can be used to call internal Angel functions from application programs. I added only the functions I needed, but the `sysInfo` array can easily be extended to

contain the address of any Angel function that is needed by an external program. The following steps have to be taken to add a function, see also the code examples in section 4.2 “Support code” for details:

- add a pointer to the function to `sysInfo` at the end of file `Angel\Source\brutus\devices.c`
- add a `#define “index”` in file `util\util_misc.h`
- add a `typedef` declaring a function pointer of that type in file `util\util_misc.h`
- add `extern function_type function_pointer` in file `util\util_misc.h`
- add a variable `function_type function_pointer` in file `util\util_misc.c`
- add an assignment in `misc_InitAngelFunctions()` at the end of file `util\util_misc.c`

Another extension I made concerns the caches. When I tried to do the SIR modulation in software (See section 5.2.2 “SIR via Software Modulation”) it turned out that the maximum toggle frequency for the pin generating the signal was far below the required 200 kHz, despite the fact that the SA-1100 is a 200 MHz RISC processor which should be easily capable of this task. Further examination showed that in accordance to the power up message displayed by the Debug Monitor the caches were enabled, but the cacheable bit was **not set** for any memory region, thus effectively disabling the caches.

I added an additional configuration flag (`CACHEABLE_BIT_ENABLED`) to the file `\Arm211\Angel\Source\brutus\devconf.h` which contains the hardware description of the system for which Angel is to be built. This define is evaluated in `\Arm211\Angel\Source\brutus\banner.h` (to add an appropriate string to the boot message), in `\Arm211\Angel\Source\brutus\makelo.c` (generates an assembler include file out of the C-configuration file) and most importantly, in `\Arm211\Angel\Source\brutus\target.s`. Depending on the value of the flag, SRAM and FLASH memory are marked as cacheable or non-cacheable in the `SA_INITMMU` macro in this file.

Another problem was related to the interrupt handling. After extensive search and disassembling it turned out that the SmartBadge I was working with contained a version of Angel that handled only three specific interrupt requests while neglecting all others. The available source code version of Angel did not have this constraint which made finding the error quite difficult. In addition it turned out that the existing code contained an error - the low level interrupt handler did not take into account the contents of the interrupt controller mask register (See section 4.3 “Interrupts” for more details), thus handling even masked interrupts. I added code to make sure that only unmasked interrupt requests are further handled by the interrupt handler. This code is also located in `\Arm211\Angel\Source\brutus\target.s.`

When experimenting with the GPIO pins I was surprised to find that the available GPIO pins GPIO2-GPIO9 were configured as outputs after reset. Further search in Angel showed that they were configured as outputs in a macro called `INIT_GPIOS` in `\Arm211\Angel\Source\brutus\target.s`. This might be a relict from the porting process were the GPIO pins were heavily used for debug purposes. In normal usage this is not desired though (power consumption and possible short circuits as discussed in section 4.4 “General Purpose I/O Controller (GPIO)” and section 4.5 “Peripheral Pin Controller (PPC)”), therefore I changed the macro to have all pins be configured as inputs. Additionally I added some code to **disable** rising and falling edge detection for these pins which was not done by default.

While most of these changes, although driven by needs resulting from my IrDA implementation, can be well used in a general Angel image I applied some more changes which are primarily meant for use in combination with the IrDA implementation. They are controlled by flags in the `devconf.h` file (mainly `IRLAN_SUPPORTED`) and described in more detail in Chapter 9. ”IrDA Implementation”.

4.1.2 Rebuilding Angel for the Badge

To rebuild an Angel image containing my changes the following steps have to be followed:

1. Install ARM Software Development Tools Version 2.11 in C:\ARM211.

Note: It must be Version 2.11, currently (Badge-) Angel cannot be rebuilt using the new ARM SDK Version 2.5 ! Also there seems to be problem if the ARM SDK is not installed into C:\ARM211.

2. Unpack the file ChwolfAngel.zip containing my Angel source code into the directory C:\ARM211\Angel\Source (if step one has been omitted because ARM SDK is already installed, any modified files in the source directory should be backed up before unpacking my file).
3. Start the ARM Project Manager
4. To rebuild standard Angel open the project `angelsa.apj`, to rebuild Angel with support for the IrDA stack open the project `angeleth.apj`, both in directory `C:\ARM211\Angel\Source\brutus.b\apm.`
5. Edit the file `C:\ARM211\Angel\Source\brutus\devconf.h` to configure Angel to your needs (mainly the two flags which I added, CACHEABLE_BIT_SUPPORTED and IRLAN_SUPPORTED and, in case of IRLAN_SUPPORTED=1, the flag BOOTP, might be of interest).
6. Rebuild the target “Bootloader image”. This will produce quite a number of warnings which can be ignored. At the end it should say: “Project up to date”. In this case it will have produced the file `angelsa.axf` (the image file name is the same for both projects, `angelsa.apj` and `angeleth.apj`) in the directory `C:\ARM211\Angel\source\brutus.b\apm\Bootloader image`.
7. The produced image file `angelsa.axf` can then be downloaded to the FLASH via “`loadfl angelsa.axf`”.

4.2 Support code

Here I present some code which utilizes the previously described extension to Angel to support calling of some of the internal Angel functions as well as some other declarations and functions which were needed during my work, but are not directly related to one of the units described in the following sections.These are the corresponding functions and global variables:

```
/*
 *      Description:
 *          various helper macros and definitions
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 */

#ifndef util_misc_h
#define util_misc_h

#ifdef IRDA
#include <irda/socket.h>
#include <irda/address.h>
#endif

#define BIT0                0x00000001
#define BIT1                0x00000002
#define BIT2                0x00000004
#define BIT3                0x00000008
#define BIT4                0x00000010
#define BIT5                0x00000020
#define BIT6                0x00000040
#define BIT7                0x00000080
#define BIT8                0x00000100
#define BIT9                0x00000200
#define BIT10               0x00000400
#define BIT11               0x00000800
#define BIT12               0x00001000
#define BIT13               0x00002000
#define BIT14               0x00004000
#define BIT15               0x00008000
#define BIT16               0x00010000
#define BIT17               0x00020000
```

Listing 1. General support declarations and macros [file util\util_misc.h]

```
#define BIT18              0x00040000
#define BIT19              0x00080000
#define BIT20              0x00100000
#define BIT21              0x00200000
#define BIT22              0x00400000
#define BIT23              0x00800000
#define BIT24              0x01000000
#define BIT25              0x02000000
#define BIT26              0x04000000
#define BIT27              0x08000000
#define BIT28              0x10000000
#define BIT29              0x20000000
#define BIT30              0x40000000
#define BIT31              0x80000000


// control bits in the program status register
// can be used to globally enable/disable IRQs and FIQs
#define MISC_F_MASK        BIT6
#define MISC_I_MASK        BIT7
#define MISC_INT_MASK      BIT7 | BIT6


// bits in register 1 of coprocessor 15 to enable/disable
// MMU, data and instruction cache and the Write Buffer
#define MISC_MMU           BIT0
#define MISC_D_CACHE       BIT2
#define MISC_WRITE_BUFFER  BIT3
#define MISC_I_CACHE       BIT12


/* two heavily used macros to access peripheral registers and to
 * test a certain bit in a register
 */
#define REG(base,offs) (*(volatile unsigned int*)(base+offs))
#define TEST_BIT(base,offs,bit) ((*(volatile unsigned int*)(base+offs)) & bit)

/* some often used type declarations */
typedef unsigned int UI32 ;
typedef unsigned short UI16;
typedef unsigned char UC8;
typedef UI32 BOOL;
#define bool BOOL


/* The Angel software trap prototype*/
int _syscall(int, int *);
#define angel_SWIreason_EnterSVC 0x17



/*
 * Function: misc_SysEnableICache
 *  Purpose: Enable the SA-1100 Instruction Cache
 *
 * Parameters: none
 *    Returns: void
 */
void misc_SysEnableICache(void);


/*
 * Function: misc_SysDisableICache
 *  Purpose: Disable the SA-1100 Instruction Cache
 *
```

Listing 1. General support declarations and macros [file util\util_misc.h]

```
 * Parameters: none
 *    Returns: void
 */
void misc_SysDisableICache(void);


/*
 * Function: misc_SysEnableDCache
 *  Purpose: Enable the SA-1100 Data Cache
 *
 * Parameters: none
 *    Returns: void
 */
void misc_SysEnableDCache(void);


/*
 * Function: misc_SysDisableDCache
 *  Purpose: Disable the SA-1100 Data Cache
 *
 * Parameters: none
 *    Returns: void
 */
void misc_SysDisableDCache(void);


/*
 * Function: misc_SysEnableWriteBuffer
 *  Purpose: Enable the SA-1100 Write Buffer (which holds data on its way to be
 *           written to external memory)
 *
 * Parameters: none
 *    Returns: void
 */
void misc_SysEnableWriteBuffer(void);


/*
 * Function: misc_SysDisableWriteBuffer
 *  Purpose: Disable the SA-1100 Write Buffer
 *
 * Parameters: none
 *    Returns: void
 */
void misc_SysDisableWriteBuffer(void);


/* for fast debugging without the real target hardware
 * In armulate the Angel extension of course does not work. Complex programs
 * needing the extension can't be debugged with Armulate anyway, but some
 * simple programs (which under Angel still require Angel_EnterSVC() /
 * Angel_ExitToUSR() were useful to debug with Armulate in which case the
 * function pointer must not be used.
 */
//#define ARMULATE
#ifdef ARMULATE
    #define Angel_EnterSVC()
    #define Angel_ExitToUSR()
#else
    #define Angel_EnterSVC()    __Angel_EnterSVC()
    #define Angel_ExitToUSR()   __Angel_ExitToUSR()
#endif


/* needed for the IrDA implementation */
```

Listing 1. General support declarations and macros [file util\util_misc.h]

```
#define cli() Angel_EnterSVC()
#define sti() Angel_ExitToUSR()
#define restore_flags(x)
#define save_flags(x)


/*
 * Function: misc_GetRandomBytes
 *  Purpose: Get one or four pseudo-random bytes
 *
 * Parameters:
 *      Input: nbytes        1 or 4 to specify the number of bytes
 *     Output: buf           the requested number of random bytes
 *    Returns: void
 *
 * This function stores either 1 or 4 pseudo-random bytes into the supplied
 * buffer (intended for the IrDA protocol). The bytes are taken from the
 * OS Timer count register. It is assumed that the supplied buffer can hold
 * the requested number of bytes.
 */
void misc_GetRandomBytes(void* buf, int nbytes);


/*
 * Function: misc_PrintErrorStdout
 *  Purpose: print an error message to the standard output
 *
 * Parameters:
 *      Input: text          The error message to output
 *    Returns: void
 *
 * Note: the function uses the semihosted function fprintf and therefore
 *       cannot be used in interrupt handlers or in queued functions !!
 */
void misc_PrintErrorStdout(char *);


/* defines and typedefs for the Angel hooks */

/* offset of first function to beginning of sysInfo array */
#define ANGEL_FUNCTION_OFFSET       5

/* indices of the currently implemented hooks */
#define ANGEL_FUNCTION_ENTER_SVC                    0
#define ANGEL_FUNCTION_EXIT_TO_USR                  1
#define ANGEL_FUNCTION_DISABLE_INTERRUPTS_FROM_SVC  2
#define ANGEL_FUNCTION_ENABLE_INTERRUPTS_FROM_SVC   3
#define ANGEL_FUNCTION_RESTORE_INTERRUPTS_FROM_SVC  4
#define ANGEL_FUNCTION_SERIALISE_TASK               5
#define ANGEL_FUNCTION_QUEUE_CALLBACK               6
#define ANGEL_FUNCTION_YIELD                        7

#ifdef IRDA
#define ANGEL_FUNCTION_NETSTART_MAIN                8
#define ANGEL_FUNCTION_CONFIGURE_IP                 9
#define ANGEL_FUNCTION_PROCESS_ONE_PACKET           10
#define ANGEL_FUNCTION_IP_GET_DEVICE_ADDRESS        11
#define ANGEL_FUNCTION_IP_SET_DEVICE_ADDRESS        12
#define ANGEL_FUNCTION_SOCKET                       13
#define ANGEL_FUNCTION_CLOSE_SOCKET                 14
#define ANGEL_FUNCTION_SENDTO                       15
#define ANGEL_FUNCTION_BIND                         16
#define ANGEL_FUNCTION_RECV_FROM                    17
#define ANGEL_FUNCTION_RECV                         18
#define ANGEL_FUNCTION_GET_SOCK_NAME                19
```

Listing 1. General support declarations and macros [file util\util_misc.h]

```
#endif


/* priority types for queued functions */
typedef enum angel_TaskPriority {
    TP_IdleLoop = 0,
    TP_AngelInit = 1,
    TP_Application = 2,
    TP_ApplCallBack = 3,
    TP_AngelCallBack = 4,
    TP_AngelWantLock = 5
} angel_TaskPriority;

#define TP_MaxEnum  (TP_AngelWantLock)

/* typedefs for queued and serialised functions */
typedef void (*angel_SerialisedFn)(void *);
typedef void (*angel_CallbackFn)(void *a1,
                                 void *a2,
                                 void *a3,
                                 void *a4);


/* typedefs for the different hook functions */
typedef void (*Angel_EnterSVC_fn)(void);
typedef void (*Angel_ExitToUSR_fn)(void);
typedef int (*Angel_DisableInterruptsFromSVC_fn)(void);
typedef int (*Angel_EnableInterruptsFromSVC_fn)(void);
typedef void (*Angel_RestoreInterruptsFromSVC_fn)(int);
typedef void (*Angel_SerialiseTask_fn)(bool called_by_yield,
                                       angel_SerialisedFn fn,
                                       void *state,
                                       unsigned empty_stack);
typedef void (*Angel_QueueCallback_fn)(angel_CallbackFn fn,
                                 angel_TaskPriority priority,
                                 void *a1, void *a2, void *a3, void *a4);
typedef void (*Angel_Yield_fn)(void);

#ifdef IRDA
typedef int (*eth_xmit_func)(unsigned char* buff, unsigned int size);

struct irlan_info_block
{
// FIXME replace with define, maybe common with Angel
    unsigned char hw_address[6];
    unsigned int* irlan_eth_xmit;
    unsigned char* balance_buf;
    unsigned char* irlan_recv_buf;
    unsigned int*  irlan_recv_buf_count;
    unsigned int*  irlan_recv_buf_size;
};

typedef int  (*Angel_NetstartMain_fn)(struct irlan_info_block* info);
typedef int  (*Angel_ConfigureIP_fn)(ip_addr ip);
typedef int  (*Angel_EthernetProcessOnePacket_fn)(int device_no,
                  unsigned char* input_buffer, unsigned char* *output_buffer);
typedef void (*Angel_IPGetDeviceAddress_fn)(int device_no, ip_addr ip);
typedef void (*Angel_IPSetDeviceAddress_fn)(int device_no,ip_addr ip);
typedef int  (*Angel_Socket_fn)(int af, int type, int protocol);
typedef int  (*Angel_Close_fn)(int socket);
typedef int  (*Angel_SendTo_fn)(int socket, const void *msg, int len, int flags,
                            const struct sockaddr *destaddr, int addrlen);

typedef int  (*Angel_Bind_fn)(int socket, struct sockaddr_in *localaddr, int addrlen);
typedef int  (*Angel_RecvFrom_fn)(int socket, void *buf, int len, int flags,
```

Listing 1. General support declarations and macros [file util\util_misc.h]

```
                              struct sockaddr *fromaddr, int *addrlen);
typedef int  (*Angel_Recv_fn)(int socket, void *buf, int len, int flags);
typedef int  (*Angel_GetSockName_fn)(int socket, struct sockaddr *localaddr, int
*addrlen);

/* the variables containing the addresses to the hook functions */
extern Angel_EnterSVC_fn __Angel_EnterSVC;
extern Angel_ExitToUSR_fn __Angel_ExitToUSR;
extern Angel_DisableInterruptsFromSVC_fn Angel_DisableInterruptsFromSVC;
extern Angel_EnableInterruptsFromSVC_fn Angel_EnableInterruptsFromSVC;
extern Angel_RestoreInterruptsFromSVC_fn Angel_RestoreInterruptsFromSVC;
extern Angel_SerialiseTask_fn Angel_SerialiseTask;
extern Angel_QueueCallback_fn Angel_QueueCallback;
extern Angel_Yield_fn Angel_Yield;
extern Angel_NetstartMain_fn Angel_NetstartMain;
extern Angel_ConfigureIP_fn  Angel_ConfigureIP;
extern Angel_EthernetProcessOnePacket_fn Angel_EthernetProcessOnePacket;
extern Angel_IPGetDeviceAddress_fn Angel_IPGetDeviceAddress;
extern Angel_IPSetDeviceAddress_fn Angel_IPSetDeviceAddress;
extern Angel_Socket_fn Angel_Socket;
extern Angel_Close_fn Angel_Close;
extern Angel_SendTo_fn Angel_SendTo;
extern Angel_Bind_fn Angel_Bind;
extern Angel_RecvFrom_fn Angel_RecvFrom;
extern Angel_Recv_fn Angel_Recv;
extern Angel_GetSockName_fn Angel_GetSockName;

#endif

/*
 * Function: misc_GetAngelFunction
 *  Purpose: Get the address of one of the hooks into Angel
 *
 * Parameters: index         The number of the function whose address is
 *                           to return
 *    Returns: The address of the requested function cast to a int pointer
 *
 * This function returns the address hook function specified by index.
 * This address then can be used to call the (internal) Angel function
 * from an application program. See the include file for the defined
 * indices.
 */
unsigned int * misc_GetAngelFunction(int index);


/*
 * Function: misc_InitAngelFunctions
 *  Purpose: Set up the global function pointers with the Angel function
 *           addresses
 *
 * Parameters: none
 *    Returns: void
 *
 * This function determines the addresses of the implemented hooks into
 * Angel and stores them in their respective global variables. The
 * function should be called first in an application program because
 * many other functions rely on these pointers. To enhance speed the
 * pointers are not checked for validity before usage.
 */
void misc_InitAngelFunctions(void);
```

Listing 1. General support declarations and macros [file util\util_misc.h]

```
/*
 *      Description:
 *          Various helper functions for use with the Badge and Angel
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#include <util/util_misc.h>
#include <util/util_ostimer.h>  // needed for the function misc_GetRandomBytes
#include <stdio.h>


/* Global variables to contain function pointers to hooks into Angel.
 * The functions are internal Angel functions which are useful for
 * application programs, but not directly accessible.
 * misc_InitAngelFunctions() sets the pointers which can then be
 * used to call the Angel functions.
 * The comments are taken from the file \arm211\Angel\Source\serlock.h
 */


/*
 *        Function:  Angel_EnterSVC
 *         Purpose:  Switch to SVC mode from USR mode, setting the I-bit
 *                   and the F-bit (two bits in the Program Status Register,
 *                   that disable IRQs and FIQs respectively, if set).
 *
 *  Pre-conditions:  The caller must presently be executing in USR mode.
 *
 *          Effect:  Execution continues in SVC transparently. The I-bit and
 *                   the F-bit are both set. A switch of stacks occurs (the
 *                   USR stack pointer and stack limit are copied to the
 *                   SVC registers), so that the transition is transparent to
 *                   APCS.
 */
Angel_EnterSVC_fn __Angel_EnterSVC;

/*
 *        Function:  Angel_ExitToUSR
 *         Purpose:  Switch back to USR mode after executing for a while
 *                   in SVC.
 *
 *  Pre-conditions:  The caller must presently be executing in SVC.
 *
 *          Effect:  Execution continues in USR mode transparently. The I-bit
 *                   and F-bit are both cleared. The SVC stack pointer and
 *                   stack limit are copied to the USR registers, so that the
 *                   transition is transparent to APCS, and the SVC stack
 *                   is reset to an empty state.
 */
Angel_ExitToUSR_fn __Angel_ExitToUSR;

/*
 *        Function:  Angel_DisableInterruptsFromSVC
 *         Purpose:  Disable interrupts while executing in SVC mode
 *
 *  Pre-conditions:  The caller must presently be executing in SVC mode
 *                   and must have obtained the serialiser lock.
 *
 *          Inputs:  None
```

Listing 2. General support functions and macros [file util_misc.c]

```
 *
 *         Returns:  Processor State on entry to the routine
 *
 *          Effect:  Interrupts are disabled.
 */
Angel_DisableInterruptsFromSVC_fn Angel_DisableInterruptsFromSVC;

/*
 *        Function:  Angel_EnableInterruptsFromSVC
 *         Purpose:  Enable interrupts while executing in SVC mode
 *
 *  Pre-conditions:  The caller must presently be executing in SVC mode
 *                   and must have obtained the serialiser lock.
 *
 *          Inputs:  Nothing
 *
 *         Returns:  Processor State on entry to the routine
 *
 *          Effect:  Interrupts are enabled.
 */
Angel_EnableInterruptsFromSVC_fn Angel_EnableInterruptsFromSVC;

/*
 *        Function:  Angel_RestoreInterruptsFromSVC
 *         Purpose:  Enable interrupts while executing in SVC mode
 *
 *  Pre-conditions:  The caller must presently be executing in SVC mode.
 *
 *          Inputs:  state      State returned from previous call to
 *                              Angel_DisableInterruptsFromSVC() or
 *                              Angel_EnableInterruptsFromSVC().
 *
 *          Effect:  The interrupt state is restored.
 */
Angel_RestoreInterruptsFromSVC_fn Angel_RestoreInterruptsFromSVC;

/*
 *        Function:  Angel_SerialiseTask
 *         Purpose:  To queue a function to be executed in a serial queue of
 *                   actions with “the lock”. In this desired state, mutual
 *                   exclusion is automatically achieved, by serialization.
 *
 *       Arguments:  called_by_yield       1 if called by Angel_Yield
 *                                         0 otherwise.
 *                   fn                    is the function which desires the lock
 *                   state                 is a parameter for fn
 *                   empty_stack           the value of the stack pointer
 *                                         of the current mode such that its
 *                                         stack is empty - it will be reset
 *                                         to this value, and hence fn must
 *                                         not need to access any items which
 *                                         might have been left there
 *
 * Implicit Argument Angel_MutexSharedTempRegBlocks[0] must hold the
 *                   interrupted regblock on entry to Angel_SerialiseTask.
 *
 *  Pre-conditions:  This function may be called from IRQ, FIQ, UND or SVC
 *                   mode.
 *
 *          Effect:  If the lock is presently unowned, fn will be executed
 *                   immediately in SVC with the I-bit and F-bit clear. If it
 *                   is already owned, however, the complete context needed
 *                   to execute it later is saved.
 *
 *                   This is not a “normal” function, in the sense that it
```

Listing 2. General support functions and macros [file util_misc.c]

```
 *                   does not necessarily preserve sequence.
 *
 *                   When fn is ready to be executed, the registers will be
 *                   as follows:
 *
 *                      r0 (a1) = <state>
 *                      sp     -> (empty) SVC stack area
 *                      sl      = SVC stack limit
 *                      fp      = 0    (no previous frames)
 *                      lr     -> entry point to NextTask
 *                      pc     -> <fn>
 *                      cpsr   => SVC mode, I-clear, F-clear
 *
 *                   Thus, when fn exits, it will invoke NextTask.
 */
Angel_SerialiseTask_fn Angel_SerialiseTask;

/*
 *        Function:  Angel_QueueCallback
 *         Purpose:  This routine enables device drivers, the breakpoint
 *                   (undefined instruction) handler, the SWI handler or the
 *                   “yield” code to queue requests.
 *
 *                   It is, in fact, just a veneer on QueueTask - for the
 *                   sake of ease of use.
 *
 *       Arguments:  fn        the function to be placed in the appropriate
 *                             Angel queue
 *                   priority  this identifies the queue into which the
 *                             request is to be placed (see the datatype
 *                             angel_TaskPriority in the include file misc_util.h).
 *                   a1        first argument to <fn> (goes in r0)
 *                   a2        second argument to <fn> (goes in r1)
 *                   a3        third argument to <fn> (goes in r2)
 *                   a4        fourth argument to <fn> (goes in r3)
 *
 *  Pre-conditions:  This code must be called in SVC mode, with the lock
 *                   (aquired via Angel_SerialiseTask).
 *
 *          Effect:  QueueTask is called, with registers set up as above,
 *                   and in addition:
 *
 *                     fp   = 0  (no previous frames)
 *                     lr  -> entry point of NextTask
 *                     pc  -> <fn>
 *                     cpsr = USR, I-clear, F-clear
 *                     sl, sp are set to be the Application stack
 *
 *            Note:  This routine is not atomic, but it does call QueueTask,
 *                   which is.
 */
Angel_QueueCallback_fn Angel_QueueCallback;

/*
 *        Function:  Angel_Yield
 *         Purpose:  This is a voluntary yield function, which allows the
 *                   polling loop to execute; this permits an application to
 *                   give control to the polling loop and perform any necessary
 *                   polling actions.
 *
 *  Pre-conditions:  This routine may be called either from USR or from SVC.
 *                   In the latter case, the lock should be held.
 *
 *          Effect:  If not in SVC, a transparent stack switch is made (the
 *                   USR stack pointer and stack limit register are copied to
```

Listing 2. General support functions and macros [file util_misc.c]

```
 *                   the SVC ones) so that the code is APCS (ARM Procedure Calls
 *                   Standard) conformant. The polling loop is then called so that
 *                   any polled devices may be checked (AngelYield()->
 *                   Angel_YieldCore()->Angel_DeviceYield() { check all polling
 *                   handlers in the polling table }.
 *
 *                   On exit, a transparent switch is made back to USR if the
 *                   original call came from USR - in which case the SVC stack
 *                   is also reset so that it is empty.
 *                   Angel_Yield can be called in “waiting”, e.g., when
 *                   waiting for a device to become ready,...
 */
Angel_Yield_fn Angel_Yield;


/*
 *        Function:  Angel_NetstartMain [file netstart.c]
 *         Purpose:  This routine initializes the UDP/IP stack for use
 *                   with the Badge IrDA-stack
 *
 *       Arguments:  irlan_info pointer to a structure containing the MAC
 *                   address of the IrLAN device and pointers to the the
 *                   irlan transmit and recive functions
 *
 *  Pre-conditions:  an irlan-connection must be initialized in order to
 *                   provide a valid MAC-address
 *
 */
Angel_NetstartMain_fn Angel_NetstartMain;
Angel_ConfigureIP_fn  Angel_ConfigureIP;

Angel_EthernetProcessOnePacket_fn Angel_EthernetProcessOnePacket;

Angel_IPGetDeviceAddress_fn Angel_IPGetDeviceAddress;
Angel_IPSetDeviceAddress_fn Angel_IPSetDeviceAddress;

// socket functions
Angel_Socket_fn Angel_Socket;
Angel_Close_fn Angel_Close;
Angel_SendTo_fn Angel_SendTo;
Angel_Bind_fn Angel_Bind;
Angel_RecvFrom_fn Angel_RecvFrom;
Angel_Recv_fn Angel_Recv;
Angel_GetSockName_fn Angel_GetSockName;

// a global variable to detect if the Angel function hooks have
// been initialized (by calling misc_InitAngelFunctions)
int misc_initialized = FALSE;


/*
 * Function: misc_SysEnableWriteBuffer
 *  Purpose: Enable the SA-1100 Write Buffer (which holds data on its way to be
 *           written to external memory)
 *
 * Parameters: none
 *    Returns: void
 */
void misc_SysEnableWriteBuffer(void)
{
    Angel_EnterSVC();
    __asm
    {
        MRC p15, 0, r0, c1, c0, 0
        MOV r1, #MISC_WRITE_BUFFER
```

Listing 2. General support functions and macros [file util_misc.c]

```
        ORR r0, r0, r1                  // set bit 3 in control reg 1
        MCR p15, 0, r0, c1, c0, 0
    }
    Angel_ExitToUSR();
}

/*
 * Function: misc_SysDisableWriteBuffer
 *  Purpose: Disable the SA-1100 Write Buffer
 *
 * Parameters: none
 *    Returns: void
 */
void misc_SysDisableWriteBuffer(void)
{
    Angel_EnterSVC();
    __asm
    {
        MRC p15, 0, r0, c1, c0, 0
        MOV r1, #MISC_WRITE_BUFFER

        BIC r0, r0, r1
        MCR p15, 0, r0, c1, c0, 0   // clear bit 3 in control reg 1
    }
    Angel_ExitToUSR();
}


/*
 * Function: misc_SysEnableICache
 *  Purpose: Enable the SA-1100 Instruction Cache
 *
 * Parameters: none
 *    Returns: void
 */
void misc_SysEnableICache(void)
{
    Angel_EnterSVC();
    __asm
    {
        MRC p15, 0, r0, c1, c0, 0
        MOV r1, #MISC_I_CACHE

        ORR r0, r0, r1
        MCR p15, 0, r0, c1, c0, 0   // set bit 12 in control reg 1
    }
    Angel_ExitToUSR();
}


/*
 * Function: misc_SysDisableICache
 *  Purpose: Disable the SA-1100 Instruction Cache
 *
 * Parameters: none
 *    Returns: void
 */
void misc_SysDisableICache(void)
{
    Angel_EnterSVC();
    __asm
    {
        MRC p15, 0, r0, c1, c0, 0
        MOV r1, #MISC_I_CACHE
```

Listing 2. General support functions and macros [file util_misc.c]

```
        BIC r0, r0, r1
        MCR p15, 0, r0, c1, c0, 0   // clear bit 12 in control reg 1

    }
    Angel_ExitToUSR();
}


/*
 * Function: misc_SysEnableDCache
 *  Purpose: Enable the SA-1100 Data Cache
 *
 * Parameters: none
 *    Returns: void
 */
void misc_SysEnableDCache(void)
{
    Angel_EnterSVC();
    __asm
    {
        MRC p15, 0, r0, c1, c0, 0
        MOV r1, #MISC_D_CACHE

        ORR r0, r0, r1
        MCR p15, 0, r0, c1, c0, 0   // set bit 2 in control reg 1
    }
    Angel_ExitToUSR();
}


/*
 * Function: misc_SysDisableDCache
 *  Purpose: Disable the SA-1100 Data Cache
 *
 * Parameters: none
 *    Returns: void
 */
void misc_SysDisableDCache(void)
{
    Angel_EnterSVC();
    __asm
    {
        MRC p15, 0, r0, c1, c0, 0
        MOV r1, #MISC_D_CACHE

        BIC r0, r0, r1
        MCR p15, 0, r0, c1, c0, 0   // clear bit 2 in control reg 1

    }
    Angel_ExitToUSR();
}


/*
 * Function: misc_PrintErrorStdout
 *  Purpose: print an error message to the standard output
 *
 * Parameters:
 *      Input: text          The error message to output
 *    Returns: void
 *
 * Note: the function uses the semihosted function fprintf and therefore
 *       cannot be used in interrupt handlers or in queued functions !!
 */
```

Listing 2. General support functions and macros [file util_misc.c]

```
void misc_PrintErrorStdout(char * text)
{
    fprintf(stdout, “Error: %s\n”, text);
    fflush(stdout);
}


/*
 * Function: misc_GetRandomBytes
 *  Purpose: Get one or four pseudo-random bytes
 *
 * Parameters:
 *      Input: nbytes        1 or 4 to specify the number of bytes
 *     Output: buf           the requested number of random bytes
 *    Returns: void
 *
 * This function stores either 1 or 4 pseudo-random bytes into the supplied
 * buffer (intended for the IrDA protocol). The bytes are taken from the
 * OS Timer count register. It is assumed that the supplied buffer can hold
 * the requested number of bytes.
 */
void misc_GetRandomBytes(void * buf, int nbytes)
{
    if(nbytes == 1)
    {
        // take a byte from the ostimer count register
        *(UI32*)buf = REG(OSTIMER_BASE, OSCR) & 0x000000ff;
    }
    else if(nbytes == 4)
    {
        // take four bytes from the ostimer count register
        *(UI32*)buf = REG(OSTIMER_BASE, OSCR);
    }
    else
    {
        *(UI32*)buf = 0;
    }
}


/*
 * Function: misc_GetAngelFunction
 *  Purpose: Get the address of one of the hooks into Angel
 *
 * Parameters: index         The number of the function whose address is
 *                           to be returned
 *    Returns: The address of the requested function cast to a int pointer
 *
 * This function returns the address of the function specified by index.
 * This address then can be used to call the (internal) Angel function
 * from an application program. See the include file for the defined
 * indices.
 */
unsigned int * misc_GetAngelFunction(int index)
{
    int block[10];
    int ret;
    int args[1];
    unsigned int* j;

    // the extended version of _syscall returns a pointer to the sysinfo
    // structure in block[4], which then can be used to access get the
    // address of some of the internal Angel functions
    args[0] = (int) (block);
    ret = _syscall(0x16, args);
```

Listing 2. General support functions and macros [file util_misc.c]

```
    j = (unsigned int*) block[4];
    return (unsigned int *) j[ANGEL_FUNCTION_OFFSET+index];
}


/*
 * Function: misc_InitAngelFunctions
 *  Purpose: Set up the global function pointers with the Angel function
 *           addresses
 *
 * Parameters: none
 *    Returns: void
 *
 * This function determines the addresses of the implemented hooks into
 * Angel and stores them in their respective global variables. The
 * function should be called first in an application program because
 * many other functions rely on these pointers and, to enhance speed, the
 * pointers are not checked for validity before usage.
 */
void misc_InitAngelFunctions(void)
{
    int block[20];
    int ret;
    int args[1];
    unsigned int* j;

    if(!misc_initialized)
    {
    // the extended version of _syscall returns a pointer to the sysinfo
    // structure in block[4], which then can be used to access get the
    // address of some of the internal Angel functions
     args[0] = (int) (block);
     ret = _syscall(0x16, args);

     j = (unsigned int*) block[4];

     __Angel_EnterSVC =
    (Angel_EnterSVC_fn)j[ANGEL_FUNCTION_OFFSET+ANGEL_FUNCTION_ENTER_SVC];

     __Angel_ExitToUSR =
    (Angel_ExitToUSR_fn)j[ANGEL_FUNCTION_OFFSET+ANGEL_FUNCTION_EXIT_TO_USR];

     Angel_DisableInterruptsFromSVC =
      (Angel_DisableInterruptsFromSVC_fn)j[ANGEL_FUNCTION_OFFSET+
                        ANGEL_FUNCTION_DISABLE_INTERRUPTS_FROM_SVC];

     Angel_EnableInterruptsFromSVC =
      (Angel_EnableInterruptsFromSVC_fn)j[ANGEL_FUNCTION_OFFSET+
                         ANGEL_FUNCTION_ENABLE_INTERRUPTS_FROM_SVC];

     Angel_RestoreInterruptsFromSVC =
      (Angel_RestoreInterruptsFromSVC_fn)j[ANGEL_FUNCTION_OFFSET+
                        ANGEL_FUNCTION_RESTORE_INTERRUPTS_FROM_SVC];

     Angel_SerialiseTask =
      (Angel_SerialiseTask_fn)j[ANGEL_FUNCTION_OFFSET+
                                  ANGEL_FUNCTION_SERIALISE_TASK];

     Angel_QueueCallback =
      (Angel_QueueCallback_fn)j[ANGEL_FUNCTION_OFFSET+
                                  ANGEL_FUNCTION_QUEUE_CALLBACK];

     Angel_Yield=
        (Angel_Yield_fn)j[ANGEL_FUNCTION_OFFSET+ANGEL_FUNCTION_YIELD];
```

Listing 2. General support functions and macros [file util_misc.c]

```
#ifdef IRDA
      Angel_NetstartMain = (Angel_NetstartMain_fn)j[ANGEL_FUNCTION_OFFSET+
                                                    ANGEL_FUNCTION_NETSTART_MAIN];

      Angel_ConfigureIP  = (Angel_ConfigureIP_fn)j[ANGEL_FUNCTION_OFFSET+
                              ANGEL_FUNCTION_CONFIGURE_IP];

      Angel_EthernetProcessOnePacket = (Angel_EthernetProcessOnePacket_fn)
          j[ANGEL_FUNCTION_OFFSET+ANGEL_FUNCTION_PROCESS_ONE_PACKET];

      Angel_IPGetDeviceAddress = (Angel_IPGetDeviceAddress_fn)
            j[ANGEL_FUNCTION_OFFSET+ANGEL_FUNCTION_IP_GET_DEVICE_ADDRESS];

      Angel_IPSetDeviceAddress = (Angel_IPSetDeviceAddress_fn )
            j[ANGEL_FUNCTION_OFFSET+ANGEL_FUNCTION_IP_SET_DEVICE_ADDRESS];

      Angel_Socket = (Angel_Socket_fn)j[ANGEL_FUNCTION_OFFSET+ANGEL_FUNCTION_SOCKET];

      Angel_Close = (Angel_Close_fn)j[ANGEL_FUNCTION_OFFSET+
                                      ANGEL_FUNCTION_CLOSE_SOCKET];

      Angel_SendTo = (Angel_SendTo_fn)j[ANGEL_FUNCTION_OFFSET+ANGEL_FUNCTION_SENDTO];

      Angel_Bind = (Angel_Bind_fn)j[ANGEL_FUNCTION_OFFSET+ANGEL_FUNCTION_BIND];

      Angel_RecvFrom = (Angel_RecvFrom_fn)j[ANGEL_FUNCTION_OFFSET+
                                      ANGEL_FUNCTION_RECV_FROM];

      Angel_Recv = (Angel_Recv_fn)j[ANGEL_FUNCTION_OFFSET+ANGEL_FUNCTION_RECV];

      Angel_GetSockName = (Angel_GetSockName_fn)j[ANGEL_FUNCTION_OFFSET+
                                      ANGEL_FUNCTION_GET_SOCK_NAME];

#endif

      misc_initialized = TRUE;
    }
}
```

Listing 2. General support functions and macros [file util_misc.c]

4.3 Interrupts

The SA-1100 offers two types of interrupts:

- FIQ: fast interrupt request
- IRQ: interrupt request

The interrupt hierarchy is a two level structure. The first level is represented by 32 different interrupt level one sources. Each of these is represented by a bit in the Interrupt Controller IRQ Pending Register (ICIP) and the Interrupt Controller FIQ Pending Register (ICFP). For each first level interrupt source the user can choose if a request by one of the sources should lead to a IRQ or FIQ. This is done by programming the Interrupt Controller Level Register (ICLR). The sources can be masked or unmasked via the Interrupt Controller Mask Register (ICMR).

Given that the interrupt is unmasked, a set bit in one of the Pending Registers causes the corresponding handler (IRQ or the FIQ) to be executed.

The second level of the interrupt structure is represented by registers contained in the device that generates the first level interrupt request. The handler must read additional status bits to determine the exact reason for the

interrupt. Usually several second level interrupts are or'ed together to cause a first level interrupt request. The different second level interrupt sources are enabled within the source device.

In general the exact reason for an interrupt can be determined by reading two registers - first the ICIP or ICFP register, depending on the handler being called, to determine the requesting device and then a status register within this device to determine the function within the device that needs to be serviced. In idle mode the mask register is ignored and any interrupt causes the SA-1100 to exit the idle mode.

The purpose of the FIQ interrupts is to provide fast service for special devices. The FIQ vector is located at the end of the exception vector table (See [6], p. 24 for further information)

Address	Exception	Mode on entry
0x0000 0000	Reset	Supervisor
0x0000 0004	Undefined instruction	Undefined
0x0000 0008	Software interrupt	Supervisor
0x0000 000C	Abort (prefetch)	Abort
0x0000 0010	Abort (data)	Abort
0x0000 0014	not used	
0x0000 0018	IRQ	IRQ
0x0000 001C	FIQ	FIQ

Table 2. Exception vector table

This arrangement can save an otherwise necessary branch to the handler routine as the FIQ handler code can be placed directly following the exception table. A dedicated processor mode (FIQ mode), providing a separate stack and its own set of registers further enhance the FIQ performance.

Figure 9 shows the block structure of the interrupt controller:

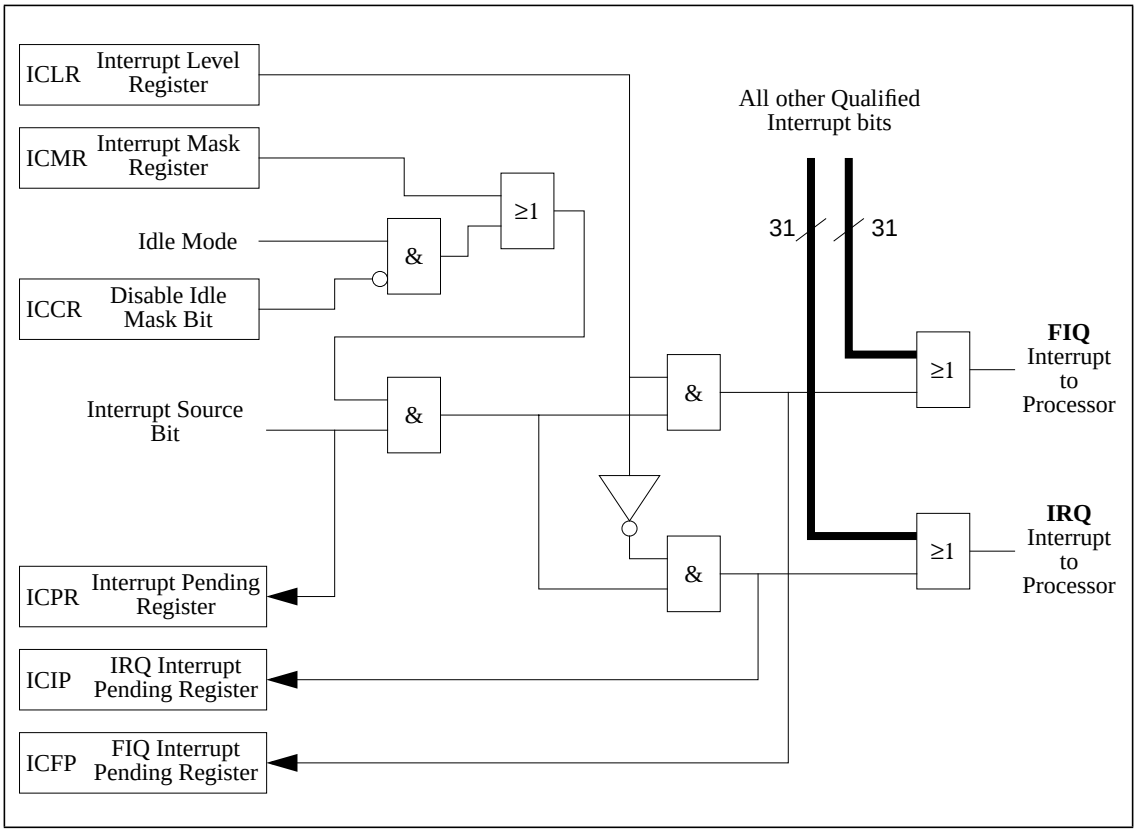


Figure 9. Interrupt Controller Block Diagram

4.3.1 Register Description

Interrupt Controller operation is determined by the following registers:

Address	Name	Description	Access
0x9005 0000	ICIP	Interrupt Controller IRQ Pending Register	RO
0x9005 0004	ICMR	Interrupt Controller Mask Register	R/W
0x9005 0008	ICLR	Interrupt Controller Level Register	R/W
0x9005 000C	ICCR	Interrupt Controller Control Register	R/W
0x9005 0010	ICFP	Interrupt Controller FIQ Pending Register	RO
0x9005 0020	ICPR	Interrupt Controller Pending Register	RO

Table 3. Interrupt Controller register block

Following reset all IRQ and FIQ interrupts are disabled and the state of all of the interrupt controller's registers is undefined. These registers must be initialized before enabling any interrupts.

4.3.1.1 Interrupt Controller Pending Register (ICPR)

The ICPR is a 32 bit read only register that shows the state of all (first-level) interrupt sources in the system. The state of this register is not affected by the mask register and does not differentiate between FIQ and IRQ requests. Table 4 shows the assignment of the different interrupt sources to the 32 bits in the ICPR. This assignment is the same for all the interrupt controller's registers. The table also lists the number of second level interrupts associated with each first level interrupt, these are or'ed together to produce the Interrupt source bit shown in Figure 9.

Bit Position	Unit	Source Module	Number of level 2 Sources	Bitfield Description
IP[31]	System	Real Time Clock	1	RTC equals alarm register
IP[30]			1	One HZ clock tick occurred
IP[29]		Operating System Timer	1	OS timer equals match register 3
IP[28]			1	OS timer equals match register 2
IP[27]			1	OS timer equals match register 1
IP[26]			1	OS timer equals match register 0
IP[25]	Peripheral	DMA Controller	3	Channel 5 service request
IP[24]			3	Channel 4 service request
IP[23]			3	Channel 3 service request
IP[22]			3	Channel 2 service request
IP[21]			3	Channel 1 service request
IP[20]			3	Channel 0 service request
IP[19]		Serial Port 4	3	SSP service request
IP[18]		Serial Port 4	8	MCP service request
IP[17]		Serial Port 3	6	UART service request
IP[16]		Serial Port 2	6+6	UART/HSSP service request
IP[15]		Serial Port 1b	6	UART service request
IP[14]		Serial Port 1a	6	SDLC service request
IP[13]		Serial Port 0	6	UDC service request
IP[12]		LCD Controller	12	LCD Controller service request

Table 4. Bits in the Interrupt Controller registers

Bit Position	Unit	Source Module	Number of level 2 Sources	Bitfield Description
IP[11]	System	General Purpose I/O	17	“OR” of GPIO Edge Detects 11..27
IP[10]			1	GPIO[10] Edge Detect
IP[9]			1	GPIO[9] Edge Detect
IP[8]			1	GPIO[8] Edge Detect
IP[7]			1	GPIO[7] Edge Detect
IP[6]			1	GPIO[6] Edge Detect
IP[5]			1	GPIO[5] Edge Detect
IP[4]			1	GPIO[4] Edge Detect
IP[3]			1	GPIO[3] Edge Detect
IP[2]			1	GPIO[2] Edge Detect
IP[1]			1	GPIO[1] Edge Detect
IP[0]			1	GPIO[0] Edge Detect

Table 4. Bits in the Interrupt Controller registers

A set bit indicates that either an FIQ or an IRQ request by that source is pending.

4.3.1.2 Interrupt Controller IRQ Pending Register (ICIP) and FIQ Pending Register (ICFP)

Both registers contain a bit per first-level interrupt source which indicates that an interrupt request has been made by one of the sources.

Usually the IRQ or FIQ service routine checks its associated pending register to determine the source unit requesting service. After having found the first level source, the software has to read this unit’s registers to service the specific function requesting interrupt service. The bits in each of the two (first level) pending registers are the result of or’ing the interrupt pending flags within the associated unit. These second level flags have to be cleared by the user in order to finally cause the first level pending bit to be cleared (done automatically by hardware), once all second level requests have been serviced. All the second level interrupt status bits are cleared by writing a one to them, while writing a zero has no effect.

4.3.1.3 Interrupt Controller Mask Register (ICMR)

The interrupt controller mask register contains one mask bit per possible (first level) source. Cleared mask bits (i.e. set to zero) prevent a pending bit from generating a processor interruption, while set bits cause an interruption if the corresponding interrupt request becomes active. As already mentioned the mask bits are ignored while the SA-1100 is in idle mode. If an interrupt request occurs while the processor is in idle mode, the interrupt becomes active regardless of the state of its mask bit (with one exception as described in section 4.3.1.5 “Interrupt Controller Control Register (ICCR)”).

The mask bits have two main applications:

- They allow software polling of interruptible sources without actually generating interrupts
- They allow the interrupt handler to prevent interrupts of lower priority from occurring while still maintaining a set of pending interrupts which may have occurred previously or occurred while servicing another interrupt.

Note that the mask register is not initialized at reset, but is cleared by Angel upon startup.

4.3.1.4 Interrupt Controller Level Register (ICLR)

The interrupt controller level register controls whether a pending interrupt request generates an FIQ or an IRQ processor interrupt. If a bit is cleared (to zero) the interrupt request is routed to the CPU’s IRQ interrupt input, if a bit

is set to one, then the request is routed to the FIQ interrupt input, both generate interrupts only if the matching interrupt is unmasked. For masked interrupts the ICLR has no effect.

4.3.1.5 Interrupt Controller Control Register (ICCR)

This register contains only a single bit, the Disable Idle Mask bit (DIM) at bit position zero. If this bit is set only unmasked interrupts can bring the SA-1100 out of idle mode. Otherwise as described before, all interrupts cause a transition out of idle mode, regardless of their masking state.

4.3.2 Interrupt handling under Angel

Initially Angel allowed only statically linked interrupt handlers, hence there was no way to use user-defined interrupt handlers with programs that are downloaded to RAM (for example, using the Debug Monitor).

On assertion of the IRQ signal the SA-1100 branches to the IRQ entry in the exception table which under Angel contains a branch to its low level interrupt handling code (found in `\Arm211\Angel\Source\suppasm.s` in a standard installation of the ARM-tools). This code switches off all other interrupts, saves the current state, sets up the stack space, and does some other necessary preparations depending on the mode the processor was in at the moment of interruption. Then it calls the macro `GETSOURCE` (in file `\Arm211\Angel\Source\brutus\target.s`) to determine, which of the 32 possible first level interrupt sources requested service (i.e. it looks for the highest set bit in the ICPR). The macro returns the position of this bit which is then used to compute an offset into a table containing the actual interrupt handlers for the various interrupt sources. The appropriate handler then is executed.

As described in 4.1 “Extensions to Angel”, Prof. G.Maguire and M.T.Smith have extended `_syscall 0x16` to return the base address of the interrupt handler table to be able to install and remove interrupt handlers at runtime. I used this modification to write some functions that allow convenient dealing with interrupt handlers under Angel.

4.3.3 Interrupt Controller Declarations and Functions

Some general macros and declarations that can be used when dealing with interrupts on the SmartBadge:

```
/*
 *      Description:
 *          Type declarations for use with interrupts on the Badge
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#ifndef util_interrupt_h
#define util_interrupt_h

#define INT_NUM_INT_HANDLERS 32

/* All SA-1100 interrupts */
#define INT_GPIO0               0
#define INT_GPIO1               1
#define INT_GPIO2               2
#define INT_GPIO3               3
#define INT_GPIO4               4
#define INT_GPIO5               5
#define INT_GPIO6               6
#define INT_GPIO7               7
#define INT_GPIO8               8
#define INT_GPIO9               9
#define INT_GPIO10              10
#define INT_GPIO_GROUP          11
#define INT_LCD                 12
```

Listing 3. Generic interrupt declarations and macros [file util\util_interrupt.h]

```
#define INT_UDC                 13
#define INT_SDLC                14
#define INT_UART1               15
#define INT_UART2               16
#define INT_UART3               17
#define INT_MCP                 18
#define INT_SSP                 19
#define INT_DMA0                20
#define INT_DMA1                21
#define INT_DMA2                22
#define INT_DMA3                23
#define INT_DMA4                24
#define INT_DMA5                25
#define INT_OSTIMER_0           26
#define INT_OSTIMER_1           27
#define INT_OSTIMER_2           28
#define INT_OSTIMER_3           29
#define INT_RTC_ONE_HERTZ       30
#define INT_RTC_ALARM           31


/*
 * The base address of the interrupt controller and the
 * offsets of the interrupt controller registers.
 */
#define INT_BASE        0x90050000

#define ICIP    0x00    // Interrupt Controller IRQ Pending Register
#define ICMR    0x04    // Interrupt Controller Mask Register
#define ICLR    0x08    // Interrupt Controller Level Register
#define ICCR    0c0C    // Interrupt Controller Control Register
#define ICFP    0x10    // Interrupt Controller FIQ Pending Register
#define ICPR    0x20    // Interrupt Controller Pending Register


/* Mask or unmask the given interrupt */
#define INT_MASK(n)             (REG(INT_BASE,ICMR) &= ~(1<<(n)))
#define INT_UNMASK(n)           (REG(INT_BASE,ICMR) |= (1<<(n)))
#define INT_IS_MASKED_Q(n)      (!(REG(INT_BASE, ICMR) & (1<<(n))))
#define INT_IS_UNMASKED_Q(n)    (REG(INT_BASE, ICMR) & (1<<(n)))

/* check for pending interrupts */
#define INT_IS_IRQ_PENDING_Q(n) (REG(INT_BASE, ICIP) & (1<<(n)))
#define INT_IS_FIQ_PENDING_Q(n) (REG(INT_BASE, ICFP) & (1<<(n)))
#define INT_IS_PENDING_Q(n)     (REG(INT_BASE, ICPR) & (1<<(n)))

/* set and get interrupt level (IRQ or FIQ) */
#define INT_SET_IRQ(n)          (REG(INT_BASE, ICLR) &= ~(1<<(n)))
#define INT_SET_FIQ(n)          (REG(INT_BASE, ICLR) |= (1<<(n)))
#define INT_GET_LEVEL(n)        (REG(INT_BASE, ICLR) & (1<<(n)))

/* set idle mode for interrupts (INT_SET_IDLE_ALL => all interrupt requests
 * will bring SA-1100 out of idle mode; INT_SET_IDLE_UNMASKED => only unmasked
 * interrupt requests will bring it out of idle mode
 */
#define INT_SET_IDLE_ALL        (REG(INT_BASE, ICCR) = 0)
#define INT_SET_IDLE_UNMASKED   (REG(INT_BASE, ICCR) = 1)



#endif
```

Listing 3. Generic interrupt declarations and macros [file util\util_interrupt.h]

Some functions to use interrupts within Angel:

```
/*
 *      Description:
 *          Code to deal with interrupt handlers on the Badge.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */


#include <stdio.h>
#include <util/util_interrupt.h>
#include <util/util_debug.h>


/* Angel interrupt handler function */
typedef void (*angel_IntHandlerFn)(
    unsigned  ident,        /* HW-specific identifier, i.e. interrupt number */
    unsigned  data,         /* data in interrupt table */
    unsigned  empty_stack   /* to be passed thru to SerialiseTask */
);


/* Angel interrupt handler entry */
typedef struct
{
  angel_IntHandlerFn    handler; /* the handler function */
  unsigned              data;    /* data passed to handler */
} angel_IntHandlerEntry;



// checked in int_Init because debug functionality
// is used in the int_DummyHandler
extern int debug_initialized;


/*
 * Function: int_DummyHandler
 *  Purpose: Default interrupt handler.
 *
 * Parameters:
 *      Input: ident              interrupt ID
 *             data               data from the angel interrupt table
 *             empty_stack        to be passed to Angel_QueueCallback
 *    Returns: void
 *
 * Dummy interrupt handler. The handler masks the interrupt that called
 * the handler and outputs a warning message on the debug UART channel.
 * This handler is intended to prevent crashes if by accident an interrupt
 * is unmasked without having set an appropriate handler that masks/handles
 * this interrupt.
 *
 * ! The dummy handler requires debug_Init() to have been called !
 * This is done automatically in int_Init(), but has to be done extra if
 * int_DummyHandler is used without having called int_Init().
 */
void int_DummyHandler(unsigned int ident, unsigned int data, unsigned int empty_stack)
{
    char buf[50];
    INT_MASK(ident);
    sprintf(buf, "\r\n\n!! unhandled interrupt %d shut down !!\n\n\r", ident);
```

Listing 4. Code to deal with interrupts under Angel [file util_interrupt.c]

```
    debug_String(buf);
}


/*
 * Function: int_Init
 *  Purpose: Set default interrupt handlers.
 *
 * Parameters: none
 *    Returns: void
 *
 * Install the error reporting dummy interrupt handler for every interrupt
 * except interrupt 17 = UART3 interrupt (this is the Angel interrupt).
 * This function should be called at the beginning of programs to prevent
 * overwriting already setup special interrupt handlers.
 */
void int_Init(void)
{
    int i;

    if(!debug_initialized)
    {
        debug_Init();
    }

    for(i=0;i<17;i++)
    {
        int_InstallHandler(i, int_DummyHandler);
    }

    for(i=18;i<INT_NUM_INT_HANDLERS;i++)
    {
        int_InstallHandler(i, int_DummyHandler);
    }
}


/*
 * Function: int_GetIntTableBase
 *  Purpose: Return the base address of the Angel interrupt handler array
 *           which then can be used to set new interrupt handlers.
 *
 * Parameters: none
 *    Returns: angel_IntHandlerEntry*  the base address of the Angel interrupt
 *                                     handler array
 *
 * By default Angel only supports statically linked interrupt handlers.
 * Prof. G.Maguire and M.T.Smith modified Angel syscall 0x16 to access the
 * interrupt handler structures:
 * Added system information for IRQs:
 *
 *   unsigned int * sysInfo[] = {
 *       (unsigned int *)angel_Device,
 *       (unsigned int*)angel_DeviceStatus,
 *       (unsigned int*)angel_IntHandler,
 *       (unsigned int*)angel_PollHandler,
 *       0
 *   };
 */
angel_IntHandlerEntry* int_GetIntTableBase(void)
{
    int block[10];
    int ret;
    int args[1];
    unsigned int* j;
```

Listing 4. Code to deal with interrupts under Angel [file util_interrupt.c]

```
    extern int _syscall(int, int *);

    // the extended version of _syscall returns a pointer to the sysinfo
    // structure in block[4], which then can be used to access the int
    // handler array.
    args[0] = (int) (block);
    ret = _syscall(0x16, args);

    j = (unsigned int*) block[4];
    return ((angel_IntHandlerEntry *) j[2]);
}


/*
 * Function: int_InstallHandler
 *  Purpose: Install a new IRQ interrupt handler
 *
 * Parameters:
 *      Input: int_nr             the interrupt number for which the handler
 *                                is to be set
 *             new_handler        the new handler function
 *
 *    Returns: the address of the old interrupt handler
 *
 */
angel_IntHandlerFn int_InstallHandler(int int_nr, angel_IntHandlerFn new_handler)
{
    angel_IntHandlerFn old_handler;
    angel_IntHandlerEntry * angel_IntHandlerArray;

    angel_IntHandlerArray = int_GetIntTableBase();
    old_handler = angel_IntHandlerArray[int_nr].handler;
    angel_IntHandlerArray[int_nr].handler =  new_handler;

    return old_handler;
}


/*
 * Function: int_PrintHandlerTable
 *  Purpose: Print the addresses of the currently set interrupt handlers
 *           to stdout.
 *
 * Parameters: none
 *    Returns: void
 *
 */
void int_PrintHandlerTable(void)
{
    int i;
    angel_IntHandlerEntry * angel_IntHandlerArray, *ientry;

    angel_IntHandlerArray = int_GetIntTableBase();

    for ( i = 0; i < INT_NUM_INT_HANDLERS; ++i ) {
        ientry = &angel_IntHandlerArray[i];
        fprintf(stdout, “entry = %d:\t”, i);
        fprintf(stdout, “handler = 0x%x\t”, (int)(ientry->handler));
        fprintf(stdout, “data = 0x%x\n”, ientry->data); }
}
```

Listing 4. Code to deal with interrupts under Angel [file util_interrupt.c]

4.3.4 Interrupt Latency under Angel

I carried out some tests to determine the latency related to interrupt handling when using Angel. I interconnected two of the free GPIO pins and configured one of them as an output, the other one as an input with interrupt on rising edges. Then I measured the time between changing the pin state to generate a rising edge and the interrupt handler being executed as a consequence of the detected edge using the OS Timer Count Register (for more information on GPIO and the OS Timer unit see sections 4.4 and 4.5 respectively). The basic measurement (measuring the time between setting a pin and execution of the interrupt handler) was repeated 15 times in a loop. This was done with four different setups (debug / release version, each of these with and without using printf, as explained further down). To determine the influence of the caches, each of the four setups was tested with the four different possible combinations of data and instruction cache enabled/disabled. The code can be found in Section B.2, Table 5 shows the results:

			Debug		Release	
			IC		IC	
			on	off	on	off
with printf	DC	on	60/51	59/39	33	29
		off	26	28	16	16
without printf	DC	on	49/4	39,21	13/4	10,28
		off	36/8	28	16/6,7	16

Table 5. Interrupt latency depending on various cache and build options.

If values are given in the format x/y, then x represents the time for the first measurement, y the times for the following ones. Values in the format x,y mean that both x and y appeared about equally frequent. The unit of the shown values is OS Timer ticks, i.e. one unit is equal to 270ns.

In the first version I used the Debug target in the ARM Project manager and output the result directly following each single measurement using printf. The measured values are shown in the table “Debug target with printf”. To my surprise the variant with both caches enabled yielded the worst results. **Disabling the instruction cache** left the time for the first loop nearly equal while the following times show significant improvement.

Disabling the data cache decreases the time by more than half with only a slight difference between instruction cache enabled and disabled. Searching for an explanation of this behaviour led to the question of could the printf-calls be responsible for the low cache efficiency, as the rest of the code in this example should easily fit into the cache. To check this out I changed the original version to only store the acquired values into an array within the loop and output them after all measurements have been executed. The results for this setup are shown in the table “Debug target without printf”. In fact nearly all values (except the first loop with IC enabled and DC disabled, and as expected the version with both caches disabled) show improvements, some of them dramatic.

Both caches enabled: The first iteration of the loop shows an improvement of more than 10% which is surprising on the first glance as the time for initially loading the caches shouldn’t differ much. A possible explanation is related to the cache allocation strategy: as the caches always allocate complete cache lines (i.e. 8 words) even a small change in the code may lead to a difference (plus or minus) of up to seven additional instructions that have to be loaded on a cache miss. The execution times for the following iterations are reduced by a factor of more than 10. Disabling the instruction cache further reduces the time for the first iteration. Both values show an improvement compared to the printf variant, but the value for the following iterations now is significantly worse than with the instruction cache enabled.

Data cache disabled, instruction cache enabled: this again shows an improvement for the first iteration over the version with the data cache enabled (though not as large as in the printf variant) and similarly the second value is much better than the first value, but now slightly worse than with the data cache enabled.

The two variants having been built with the Release Target generally show the same behaviour, but are (partly significantly) faster in nearly every measurement. However, there are no differences between the first and the following iterations in the Release printf variant.

4.3.4.1 Conclusions

The results show that the effectivity of the caches tremendously depends on the code - even slight changes (in either Angel or the application program that at first seem utterly without any influence on the performance of a particular section) can completely change the behaviour and timing - merely changing the position of the time critical code by inserting a few instructions in some **other** part of the program can make all the difference.

The fact that the performance is greatly increased if the printf statements are taken out of the loop (although none of them is in the time critical path) shows that the printf code fills up a large percentage of the cache, hence replacing most of the other code. This is further backed up by the fact that the version with both caches disabled is among the fastest - for every iteration of the loop (most of) the cache has to be reloaded from external memory - not just single instructions but complete cachelines of eight words which in this particular case seems to load additional, not needed, code which results in a significant performance penalty.

This might also explain why disabling the data cache in some cases improves the performance - only a few data locations are used but complete cache lines have to be loaded all the time. This can be as bad as having to load two cachelines of eight words each to use only two words if they are located across a cache line border.

If reproducible and short interrupt latency is required it might be worth trying to configure the memory block that contains the exception vector table as uncacheable (via the MMU): on each CPU interrupt request the pipeline is stalled and the processor branches to entry 7 of the exception table and executes the branch to the low level interrupt handler which is stored there. If the instruction cache is enabled this results in a load of eight words instead of only the one which is actually needed. This might not matter too much if it occurs only the first time and then the code executes even faster out of the cache, but if other code in the meantime replaces this cacheline, interrupt latency suffers a penalty. No general recommendation can be given, for each application this has to be determined separately using the final code.

To sum up, the caches can greatly enhance the performance (and in fact were necessary to be able to do the infrared SIR modulation in software (section 5.2.2) but can also lead to heavy performance penalties if critical code segments map to the same cache locations. Even tiny changes to the code in one location can lead to a completely changed behaviour and a performance change in other parts of the program.

Note: in the current Angel implementation the low level interrupt handler (actually the macro GETSOURCE) determines the interrupt request to service through the following mechanism: starting with bit 0 of the ICIP register it tests each bit up to bit 31 (except for bit 17 - the interrupt request bit for UART3, which is used as the Angel communication port, e.g. for debugger communication - which is tested at the end). The last set bit in this chain of tests “survives” and is considered as the highest priority request to be served. This implementation results in constant latency for all interrupts but causes unnecessary delays for the higher prioritized interrupts. If the test sequence were reversed - starting with the highest priority bit and in the case of a set bit immediately branching to the code that sets up everything before calling the high level interrupt handler for this source, thus skipping the tests for the lower prioritized interrupts - the higher prioritized interrupts could be accelerated. The cost would be an additional conditional (i.e., it has to be loaded in any case but will only be executed if the condition is passed, i.e., the bit is actually set) branch instruction per first level interrupt source. Again depending on the particular application, this might be tolerable and preferable over the current implementation.

4.4 General Purpose I/O Controller (GPIO)

The SA-1100 provides 28 general purpose I/O port pins for application specific digital input and output. Each of these pins can be programmed to act as an input or output and optionally to trigger an interrupt on detection of a rising and/or falling edge. Additionally most of the pins can be used to provide alternate functions for certain modes within the serial controllers and the LCD controller that require extra pins. Following a hard reset all pins are configured as inputs. Configuring them as inputs prevents short circuits and reduces the processor’s power consumption as there is no need to drive these pins - it may however increase overall system power consumption (See 4.5.1.4 for further details). Therefore actual output pins should be configured as outputs as soon as possible after reset.

Figure 10 shows a block diagram of a single GPIO pin. The different registers and their function is described afterwards. For all the registers bits 0 to 27 correspond to the 28 GPIO pins 0 to 27, bits 28 to 31 are reserved and return zero on reads, while writes are ignored.

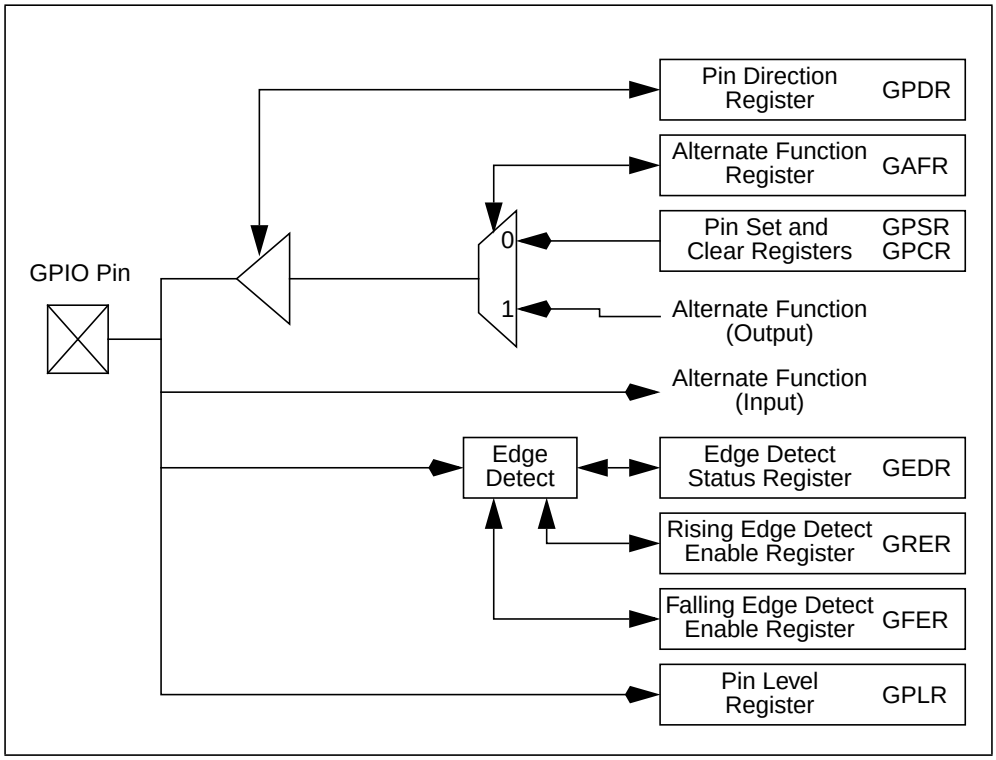


Figure 10. GPIO port block diagram

4.4.1 Register Description

The GPIO block is controlled by the following eight registers:

Address	Name	Description	Access
0x9004 0000	GPLR	GPIO Pin Level Register	RO
0x9004 0004	GPDR	GPIO Pin Direction Register	R/W
0x9004 0008	GPSR	GPIO Pin Output Set Register	WO
0x9004 000C	GPCR	GPIO Pin Output Clear Register	WO
0x9004 0010	GRER	GPIO Rising-Edge Detect Register	R/W
0x9004 0014	GFER	GPIO Falling-Edge Detect Register	R/W
0x9004 0018	GEDR	GPIO Edge Detect Status Register	R/W
0x9004 001C	GAFR	GPIO Alternate Function Register	R/W

Table 6. GPIO register block

4.4.1.1 GPIO Pin Level Register (GPLR)

The GPIO Pin Level Register monitors the state of each of the GPIO pins. Bits 0 to 27 correspond to the 28 GPIO pins and each one shows the current level of the associated pin, **regardless** of the programmed pin direction. The register is read-only. As the register shows the current state of the pins regardless of the pin direction it can also be used to detect conflicts with external hardware (e.g. if external hardware drives the pin to a different value than programmed via the GPSR/GPCR registers).

4.4.1.2 GPIO Pin Direction Register (GPDR)

This register controls whether a pin acts as an input or output pin. Writing a one to one of the GPDR bits makes the corresponding GPIO pin an output pin. Programming a bit to zero makes the GPIO pin an input. As mentioned before, following a hard reset all bits in this register are cleared, thus configuring all GPIO pins as inputs. Soft resets and sleep resets have no effect on the GPDR register.

4.4.1.3 GPIO Output Set Register (GPSR) and Output Clear Register (GPCR)

When a GPIO pin is configured as an output the state of the pin can be controlled by writing to the output pin set register (GPSR) or the output pin clear register (GPCR). To set a pin, a one has to be written to the corresponding bit within the GPSR, to clear a pin a one has to be written to its corresponding bit within the GPCR. These two registers are write only registers, reads return unpredictable values. Writing a zero to any of the GPSR or GPCR bits has no effect, as well as writing a one to a GPSR or GPCR bit corresponding to a pin which has been configured as an input. (However, the programmed values will set the pin state if the pin is configured as an output later on.)

4.4.1.4 GPIO Rising Edge Detect Register (GRER) and Falling Edge Detect Register (GFER)

Each GPIO port can be programmed to detect a rising-edge, falling edge, or either transition of a pin. When an edge is detected which matches the type of edge programmed for the pin, a status bit is set. The interrupt controller can be programmed to signal an interrupt to the CPU or wakeup the SA-1100 from sleep mode when any one of these status bits is set.

Writing a one to a bit in the Rising Edge Detect Register (GRER) causes the corresponding bit in the GPIO Edge Detect Status Register (GEDR) to be set, if a rising edge (i.e. a transition from logic level zero to one) is detected on the corresponding port pin. Likewise writing a one to the Falling Edge Detect Register (GFER) causes the bit in the GEDR to be set if a falling edge is detected on the pin. If the bits are set in both registers any transition on the port causes the status bit to be set.

4.4.1.5 GPIO Edge Detect Status Register (GEDR)

When an edge detect occurs on a pin that matches the type of edge programmed in the GRER and/or GFER registers, the corresponding status bit in GEDR is set. Once a status bit is set the CPU must clear it by writing a one to that bit. Writing zeros to status bits has no effect. This technique is also used in some other status registers and allows single write operations instead of read-modify-write cycles (just writing a one to that particular bit instead of OR or AND the bit in).

An edge detect which sets the corresponding GEDR status bit can trigger an interrupt request. Pins 11-27 form a group that can cause an interrupt request to be triggered if any of the status bits 11 through 27 is set. The interrupt handler then has to read the GEDR register to find the pin(s) that caused the interrupt. GPIO pins 0-10 each can cause an independent first level interrupt. Enabling interrupts is described in Section 4.3.

Note: edge detection is independent of the pin direction, i.e. also changing the state of pins configured as outputs causes the edge detection bits to be set, if the corresponding edge detection type is enabled.

4.4.1.6 GPIO Alternate Function Register (GAFR)

This register contains 28 control bits which correspond to the 28 GPIO pins. When a bit is set in the GAFR, the corresponding GPIO pin is switched over to that pin’s alternate function and cannot be used as a GPIO any more. Alternate functions include: sample clock input for the serial ports, UART receive and transmit pins for serial port one (if SDLC and UART mode are to be used at the same time), etc. More information can be found in [6].

4.4.2 GPIO Declarations

The following are some declarations and macros to facilitate using GPIO functionality:

```
/*
 *      Description:
 *          Type declarations and macros for use with the GPIO functions
 *                      on the badge
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#ifndef util_gpio_h
#define util_gpio_h

#define GPIO_BASE       0x90040000      /* base address of GPIO registers */

#define GPLR    0x00    /* GPIO Pin Level Register */
#define GPDR    0x04    /* GPIO Pin Direction Register */
#define GPSR    0x08    /* GPIO Pin Output Set Register */
#define GPCR    0x0C    /* GPIO Pin Output Clear Register */
#define GRER    0x10    /* GPIO Rising-Edge Detect Register */
#define GFER    0x14    /* GPIO Falling-Edge DetectRegister */
#define GEDR    0x18    /* GPIO Edge Detect Status Register */
#define GAFR    0x1C    /* GPIO Alternate Function Register */


/* GPIO port pins */
#define GPIO0           0x00000001
#define GPIO1           0x00000002
#define GPIO2           0x00000004
#define GPIO3           0x00000008
#define GPIO4           0x00000010
#define GPIO5           0x00000020
#define GPIO6           0x00000040
#define GPIO7           0x00000080
#define GPIO8           0x00000100
#define GPIO9           0x00000200
#define GPIO10          0x00000400
#define GPIO11          0x00000800
#define GPIO12          0x00001000
#define GPIO13          0x00002000
#define GPIO14          0x00004000
#define GPIO15          0x00008000
#define GPIO16          0x00010000
#define GPIO17          0x00020000
#define GPIO18          0x00040000
#define GPIO19          0x00080000
#define GPIO20          0x00100000
#define GPIO21          0x00200000
#define GPIO22          0x00400000
#define GPIO23          0x00800000
#define GPIO24          0x01000000
#define GPIO25          0x02000000
#define GPIO26          0x04000000
#define GPIO27          0x08000000


/* macros to set pin direction, state and read current state */

// set the given GPIO pin(s) as output
```

Listing 5. GPIO declarations and macros [file util\util_gpio.h]

```
#define GPIO_SET_OUTPUT(pin)    (REG(GPIO_BASE,GPDR) |= (pin))

// set the given GPIO pin(s) as input
#define GPIO_SET_INPUT(pin)     (REG(GPIO_BASE,GPDR) &= ~(pin))

// set the given GPIO pin(s) (set it to one)
#define GPIO_SET_PIN(pin)       (REG(GPIO_BASE,GPSR) = (pin))

// clear the given GPIO pin(s) (set it to zero)
#define GPIO_CLEAR_PIN(pin)     (REG(GPIO_BASE,GPCR) = (pin))

// return the current level of the given GPIO pin(s)
#define GPIO_READ_PIN(pin)      (REG(GPIO_BASE,GPLR) & (pin))


/* macros related to edge detection */

// enable / disable rising edge detection for the given GPIO pin(s)
#define GPIO_ENABLE_RISING_EDGE(pin)    (REG(GPIO_BASE, GRER) |= (pin))
#define GPIO_DISABLE_RISING_EDGE(pin)   (REG(GPIO_BASE, GRER) &= ~(pin))

// enable / disable falling edge detection for the given GPIO pin(s)
#define GPIO_ENABLE_FALLING_EDGE(pin)   (REG(GPIO_BASE, GFER) |= (pin))
#define GPIO_DISABLE_FALLING_EDGE(pin)  (REG(GPIO_BASE, GFER) &= ~(pin))

// read/reset the edge detect status of the given GPIO pin(s)
// IMPORTANT: when an edge detect has occured, the corresponding status
// bit has to be cleared by the CPU by writing a one to it, writing zeros
// doesn't affect the register.
#define GPIO_READ_EDGE_STATUS(pin)      (REG(GPIO_BASE, GEDR) & (pin))
#define GPIO_CLEAR_EDGE_STATUS(pin)     (REG(GPIO_BASE, GEDR) = (pin))
#define GPIO_CLEAR_EDGE_STATUS_ALL      (REG(GPIO_BASE, GEDR) = 0xFFFFFFFF)


/* alternate functions */

// enable / disable alternate function for a given pin
#define GPIO_ENABLE_ALTERNATE_FUNCTION(pin)  (REG(GPIO_BASE, GAFR) |= (pin))
#define GPIO_DISABLE_ALTERNATE_FUNCTION(pin) (REG(GPIO_BASE, GAFR) &= ~(pin))
#endif
```

Listing 5. GPIO declarations and macros [file util\util_gpio.h]

4.5 Peripheral Pin Controller (PPC)

While the GPIO block is located in the System Control Module, the PPC is part of the Peripheral Control Module, but as the PPC provides similar functionality as the GPIO block I chose to describe it here.

The peripheral pin controller (PPC) takes individual control of the LCD's and serial port 1-4's pins when one or more of the units are disabled, allowing the user to utilize them as general purpose digital I/O pins. When controlled by the PPC these pins operate similarly to GPIO pins, with the exception that they cannot perform edge-detection or generate interrupt requests. An additional feature is the possibility to specify the direction of the peripherals' pins when sleep mode is entered. A further difference is that rather than two set/clear registers and a pin level register there is a single state register (PPSR)

The serial ports 1-3 contain separate enables for their transmit and receive engines. If only half-duplex operation is required, one pin can be used for the serial communication while the other pin remains free for digital I/O. The pins of serial port 0 are dedicated to the USB device controller and thus cannot be used as digital I/O pins as the USB controller drives a differential transceiver.

After a hard reset all peripheral control module units are disabled which gives control of their pins to the PPC (except for serial port 0). To prevent short circuits the PPC controller then configures these pins as inputs. Similar to the case for GPIO, if PPC pins are intended to be used as outputs, they should be configured as soon as possible.

To limit off-chip power consumption during sleep mode the PPC contains a special register that, unlike the peripherals, is not reset upon entry to sleep mode. This register allows the programmer to explicitly configure 22 of the peripherals' pins as either outputs or inputs during sleep mode. Without that register, resetting the peripherals would configure all pins as inputs. When sleep mode is exited the direction and state of the pins is maintained until a special bit in the power manager is set (Release Peripheral Pin, RPP), which should be done after the peripherals have been reprogrammed. Once RPP is set, control of the peripherals' pins is given back to the individual peripherals and to the PPC unit.

4.5.1 Register Description

The PPC contains the following registers:

Address	Name	Description	Access
0x9006 0000	PPDR	PPC Pin Direction Register	R/W
0x9006 0004	PPSR	PPC Pin State Register	R/W
0x9006 0008	PPAR	PPC Pin Assignment Register	R/W
0x9006 000C	PSDR	PPC Sleep Mode Direction Register	R/W
0x9006 0010	PPFR	PPC Pin Flag Register	RO

Table 7. PPC register block

4.5.1.1 PPC Pin Direction Register (PPDR)

The Pin Direction Register is used to set the pin direction (output or input) if the corresponding peripheral is disabled, otherwise the bit is ignored. If the direction bit is programmed to a one, the pin acts as an output, if the bit is cleared, the pin works as an input. As mentioned, following reset all peripherals are reset which gives control of their pins to the PPC which in turn configures them as inputs by clearing all bits in the Pin Direction Register.

Figure 11 shows the location of each pin direction bit and to which peripheral pin it corresponds.

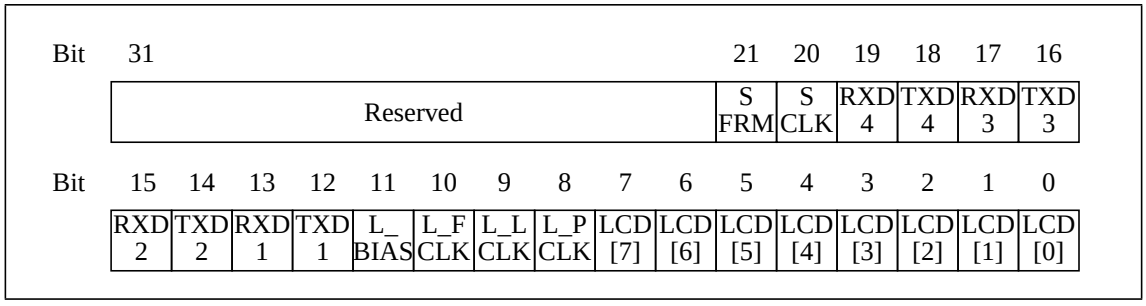


Figure 11. PPC Pin Direction Register

Bits 0-11 belong to the LCD Controller, bits 12 and 13 belong to serial port 1, bits 14 and 15 to serial port 2, bits 16 and 17 to serial port 3, and bits 18-21 to serial port 4.

4.5.1.2 PPC Pin State Register (PPSR)

Pin state of the PPC pins can be both monitored and controlled by reading/writing the PPC Pin State Register. The register contains one bit for each of the 22 peripheral pins under control of the PPC. The register can be read at any time to determine the current state of the pin, even if the pin is controlled by a peripheral rather than by the PPC.

If a peripheral is disabled and the direction of the pin is set to output (via the corresponding bit in the PPDR) its state can be controlled by writing to the Pin State Register. Writing a value to a pin state bit whose port is an input or under control of a peripheral has no effect.

PPSR is implemented as two separate registers, one of which is accessed by reads and monitors the pin state, while the other is accessed by writes and is used to control the state of pins set as GPIO outputs. The readable register is synchronized with the actual pin state at a frequency of 7.3728MHz. Therefore, after changing the state of an output pin via a write to the PPSR, it takes some time until the value is output to the pin and the change is reflected in the monitor register.

4.5.1.3 PPC Pin Assignment Register (PPAR)

Serial port 1 and serial port 4 each support two different protocols - UART/SDLC and SSP/MCP respectively. To allow both protocol engines to be used simultaneously the engines can be reassigned to GPIO pins (alternate functions) via two bits in the PPC Pin Assignment Register.

- UART Pin Reassignment (UPR)

  If UPR=0 serial port 1 uses its standard TXD1 and RXD1 pins and the SDLC/UART select (SUS) bit in SDLC control register 0 selects either UART or SDLC operation.

  If UPR=1, SUS is ignored and the SDLC engine uses pins TXD1 and RXD1, while the UART engine is connected to GPIO14 for transmission and GPIO15 for receiving. To enable proper operation bits 14 and 15 in the GPIO Alternate Function Register (GAFR) must be set and in the GPIO Pin Direction Register (GPDR) bit 14 must be set and bit 15 must be cleared to enable the alternate functionality and set the ports as output/input respectively.

- SSP Pin Reassignment (SPR)

  When SPR=0, serial port 4 uses its standard pins TXD4, RXD4, SCLK, and SFRM and the MCP enable (MCE) and SSP enable (SSE) bits are used to select the protocol for serial port 4.

  If SPR=1, MCE and SSE must both be set and serial port 4 defaults to MCP operation using the standard pins TXD4, RXD4, SCLK, and SFRM. SSP is configured to use GPIO10 for transmission, GPIO11 for receiving, GPIO12 for the serial clock, and GPIO13 for serial frame. Again bits 10-13 in GPIO GAFR must be set, as well as bits 10, 12, and 13 in GPIO GPDR while bit 11 must be cleared.

4.5.1.4 PPC Sleep Mode Pin Direction Register (PSDR)

As already mentioned above, the PPC allows the user to set the direction of the PPC pins during sleep mode to prevent external devices attached from burning power due to floating inputs. The sleep Mode Pin Direction Register contains a bit for each of the 22 pins under control of the PPC. Writing a one to a bit causes the associated pin to be configured as an input during sleep, while writing a zero to a bit causes the pin to be configured as an output which is driven low during sleep.

4.5.1.5 PPC Pin Flag Register (PPFR)

This read-only register contains eight bits which tell which peripherals are currently under control of the PPC. If a unit is enabled, the corresponding bit is cleared, if a unit is disabled (and thus the associated pin is under control of the PPC) the bit is set. As serial ports 1-3 allow separate enabling/disabling of their transmit and receive engines, separate flags exist for their transmit and receive pins.

Figure 12 shows the location of the flag bits:

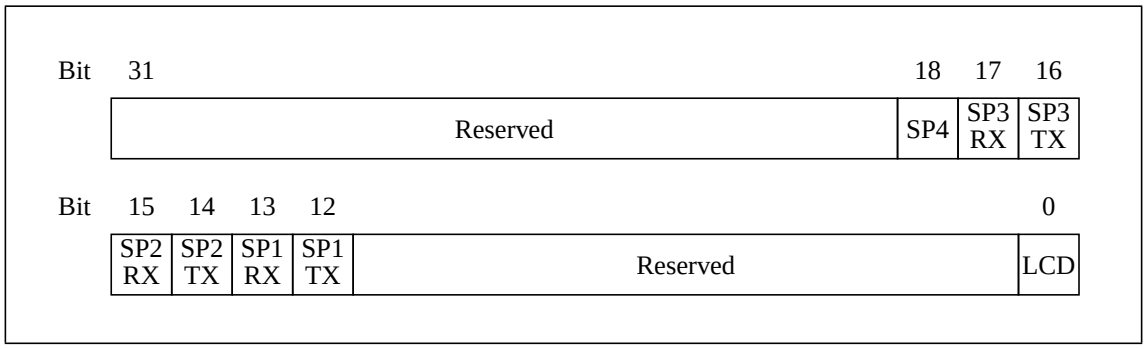


Figure 12. PPC Pin Flag Register

4.5.2 PPC declarations

Some macros and declarations to access the PPC:

```
/*
 *      Description:
 *          Type declarations and macros for use with the PPC functions
 *                      on the badge
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#ifndef util_ppc_h
#define util_ppc_h

#define PPC_BASE   0x90060000   /* base address of PPC registers */

#define PPDR    0x00            /* PPC Pin Direction Register */
#define PPSR    0x04            /* PPC Pin State Register */
#define PPAR    0x08            /* PPC Pin Assignment Register */
#define PSDR    0x0C            /* PPC Sleep Mode Direction Register */
#define PPFR    0x10            /* PPC Pin Flag Register */

/* PPC port pins */
#define LDD0            0x00000001
#define LDD1            0x00000002
#define LDD2            0x00000004
#define LDD3            0x00000008
#define LDD4            0x00000010
#define LDD5            0x00000020
#define LDD6            0x00000040
#define LDD7            0x00000080
#define L_P_CLK         0x00000100
#define L_L_CLK         0x00000200
#define L_F_CLK         0x00000400
#define L_BIAS          0x00000800
#define TXD1            0x00001000
#define RXD1            0x00002000
#define TXD2            0x00004000
#define RXD2            0x00008000
#define TXD3            0x00010000
#define RXD3            0x00020000
#define TXD4            0x00040000
#define RXD4            0x00080000
```

Listing 6. PPC declarations and macros [file util\util_ppc.h]

```
#define S_CLK           0x00100000
#define S_FRM           0x00200000


/* PPC pin reassignment flags */
#define PPC_UPR         0x00001000
#define PPC_SPR         0x00040000


/* units for the PPC_READ_FLAG macro */
#define PPC_LCD         0x00000001
#define PPC_SP1_TX      0x00001000
#define PPC_SP1_RX      0x00002000
#define PPC_SP2_TX      0x00004000
#define PPC_SP2_RX      0x00008000
#define PPC_SP3_TX      0x00010000
#define PPC_SP3_RX      0x00020000
#define PPC_SP4         0x00040000

/* macros to set pin direction, state and read current state */
#define PPC_SET_OUTPUT(pin)     (REG(PPC_BASE,PPDR) |= (pin))
#define PPC_SET_INPUT(pin)      (REG(PPC_BASE,PPDR) &= ~(pin))
#define PPC_GET_DIRECTION(pin)  (REG(PPC_BASE,PPDR) & (pin))

#define PPC_SET_PIN(pin)        (REG(PPC_BASE,PPSR) |= (pin))
#define PPC_CLEAR_PIN(pin)      (REG(PPC_BASE,PPSR) &= ~(pin))

#define PPC_READ_PIN(pin)       (REG(PPC_BASE,PPSR) & (pin))

/* enable/disable parallel operation of SDLC and UART engines for port 1
 * (pin reassignment)
 */
#define PPC_ENABLE_PORT1_DUAL_OP        (REG(PPC_BASE, PPAR) |= (PPC_UPR))
#define PPC_DISABLE_PORT1_DUAL_OP       (REG(PPC_BASE, PPAR) &= ~(PPC_UPR))

/* enable/disable parallel operation of MCP and SSP engines for port 4
 * (pin reassignment)
 */
#define PPC_ENABLE_PORT4_DUAL_OP        (REG(PPC_BASE, PPAR) |= (PPC_SPR))
#define PPC_DISABLE_PORT4_DUAL_OP       (REG(PPC_BASE, PPAR) &= ~(PPC_SPR))

/* set PPC pins as input or (low driven) output during sleep state */
#define PPC_SLEEP_INPUT(pin)    (REG(PPC_BASE, PSDR) |= (pin))
#define PPC_SLEEP_OUTPUT(pin)   (REG(PPC_BASE, PSDR) &= ~(pin))

/* determine if a unit is under control of the PPC or the corresponding
 * peripheral
 */
#define PPC_READ_FLAG(unit)     (REG(PPC_BASE, PPFR) & (unit))

#endif
```

Listing 6. PPC declarations and macros [file util\util_ppc.h]

4.5.3 Maximum Toggling Frequency of GPIO and PPC Pins

While trying to find a solution to get around the SA-1100 SIR bug with software modulation (see section 5.2 for more details), I experimented with the GPIO and PPC pins. Here I present some measurements regarding the maximal achieveable toggling frequency for GPIO and PPC pins. The measurements were done by using the two programs described in section B.1. There are two versions, one running under Angel, the other one is a standalone version to check if Angel imposes performance penalties. The standalone version was executed out of FLASH as well as out of SRAM, the Angel version could only be executed out of SRAM (I did not write a version which could be statically compiled into Angel). Both GPIO and PPC pin toggling was measured once with the instruction cache enabled and once with the instruction cache disabled (as there are no data accesses to cacheable locations the data cache has no effect here). In the case of the PPC pin toggling I added an assembler loop as an alternative because the

code generated by the compiler was far from being optimal. All three variants (GPIO, compiler generated PPC, and hand optimized PPC) executed a tight loop which just set and cleared a pin, thus generating a slightly asymmetric square wave signal. Table 8 shows the results:

	Standalone		Angel	Instruction
	FLASH	SRAM	SRAM	count
GPIO cached	50-60ns/130-140ns	50-60ns/130-140ns	60ns/120-130ns	2+0+1
no cache	700ns/2.7us	400ns/1.54us	400ns/1.5us	2+0+1
PPC cached	150ns/330ns	160ns/330ns	130ns/260ns	6+2+1
cached, opt.	150ns/310ns	150ns/310ns	130ns/260ns	4+2+1
PPC no cache	2.76us/6.88us	1.54us/3.82us	1.52us/3.76us	6+2+1
no cache, opt.	2.08us/5.54us	1.17us/3.08us	1.15us/3.03us	4+2+1

Table 8. Maximum toggling of GPIO and PPC pins [high time/period time]

The times were measured using a 100MHz HP Logic Analyzer, i.e. the finest resolution was 10ns. Deviations of +/-10ns were quite frequent in some measurements, in these cases I put both values. In the other cases a few values differed by +/-10ns, but the majority of the values were the one given in the table. The first number represents the high time of the resulting signal, the second number the whole period length (it includes the time for the branch, therefore it's not exactly twice the first number). Due to timing constraints just a small number of samples was looked at, therefore the values cannot be taken as highly exact, but they still give some clear hints:

- As expected, when executing out of cache, the FLASH and the SRAM version yield practically equal results.
- Execution out of SRAM instead of FLASH results in a performance gain by nearly a factor 2 when the cache is disabled. This is not surprising as the data path to/from SRAM is 32 bits wide while the path from the FLASH is only 16 bits wide.
- Enabling the cache speeds up by about a factor of 10 in the case of SRAM, nearly factor 20 in the case of FLASH.
- The optimized version for the PPC loop results in a performance gain of nearly 30% when the cache is disabled. With the cache enabled there is only a very small performance gain as now the sampling frequency of the PPC pin is the limiting factor instead of the delay caused by additional instructions.
- For the PPC case the Angel version with cache enabled reaches the theoretical limit which is imposed by the sampling frequency of the PPC pin state register while the GPIO pins work at a higher frequency.
- The PPC unit suffers from having only one pin state register. In general a read-modify-write cycle is necessary to change the state of a PPC pin, i.e. a load and a store operation and a data manipulation operation (OR or bit clear), giving a total of three instructions. In this special case it could have been reduced to the data manipulation operation as no other pins are involved in this test program. But in general, when the other pins' operation is not always known, the full cycle has to be undergone. The GPIO unit on the other hand has separate registers for setting and clearing pins. Thus per state change only one (write) operation is necessary. So if due to a shortage of GPIO pins one has to consider additionally using PPC pins, the decision of which function should be done via GPIO vs. PPC should be made based on the required speed and expected usage of the pin.
- The comparison between the standalone SRAM and Angel SRAM version shows a slight advantage for the Angel version. The setup for the memory timing is the same in both versions, but the Angel version runs with the MMU enabled. The reason for the difference might be related to this fact, but I did not further investigate this question.

The instruction count in the last column gives the number of load/store + manipulation + branch instructions that make up the whole loop. Disassembling the compiler generated code for the PPC showed that for each of the two state changes it reloaded the PPC register base address from one register into another, although the second register was never changed - resulting in two additional, unnecessary instructions.

Note: the measurements shown were acquired using the Release target in the ARM Project Manager. Using the Debug target resulted in completely different code which yielded far worse results. For a more detailed discussion on the code generation see section 6. “Accessing Peripherals, Code Generation with the ARM Compiler”.

4.6 Real Time Clock

The SA-1100 contains a real time clock (RTC) to provide a general purpose time reference for the system. After hardware reset the RTC is uninitialized and must be set to the desired value. Following that the counter will remain valid until the next hardware reset, which is assumed to occur infrequently. Transitions into and out of sleep mode, software reset, and watchdog reset do not affect the RTC counter. The counter is incremented on rising edges of the 1 HZ clock.

In addition to the counter (which is accessible via the RTC Count Register, RCNR) the RTC unit contains a 32 bit Alarm register (RTAR). This Alarm register can be programmed to a value which will be compared against the counter after each incrementation of the counter register. If the two registers match, a status bit is set which can be programmed to cause a first level interrupt. A second flag which also can generate an interrupt is set on each rising edge of the 1 HZ clock. Both status bits can be cleared by writing a one to their bit position.

The 1 HZ clock is generated by dividing down the 32.768 kHZ crystal oscillator output. The divider logic is programmable which allows the user to trim the clock to adjust inaccuracies in the crystal’s frequency. The trimming mechanism allows you to adjust the RTC to an accuracy of +/- 5 seconds a month. Apart from that this mechanism could also be used to generate an arbitrary frequency of up to 32.768 kHZ, in which case the clock is no longer 1 Hz!

4.6.1 Register Description

The real time clock is controlled by the following four registers:

Address	Name	Description	Access
0x9001 0000	RTAR	RTC Alarm Register	R/W
0x9001 0004	RCNR	RTC Count Register	R/W
0x9001 0008	RTTR	RTC Timer Trim Register	R/W
0x9001 0010	RTSR	RTC Status Register	R/W

Table 9. Real Time Clock register block

4.6.1.1 RTC Counter Register (RCNR)

The RTC counter register is a read/write register and is not cleared by any reset source. The counter may be written at any time, although write access should be restricted to the OS via the MMU protection mechanisms.

Writes to the register are controlled by a hardware mechanism that delays the actual write by up to one 32 KHz clock (i.e. ~30us) after the processor store to compensate for the asynchronous nature of the 1 Hz clock relative to the processor clock.

After a processor write to the RCNR all other writes to this register location are ignored until the new value has actually been loaded into the counter. The RCNR may be read at any time, reads reflect the value in the counter immediately after increment or load.

4.6.1.2 RTC Alarm Register (RTAR)

The Real Time Clock Alarm Register is a 32 bit read/write register. Following each rising edge of the 1 Hz clock, this register is compared to the RCNR. If both values are equal, then the Alarm bit in the status register is set. If in addition the Alarm Interrupt Enable bit is set this causes a first level interrupt.

After a hardware reset this register is undefined.

4.6.1.3 RTC Status Register (RTSR)

The status register contains four valid bits (see Figure 13), all other bits are reserved and return zeros on read and are unaffected by writes.

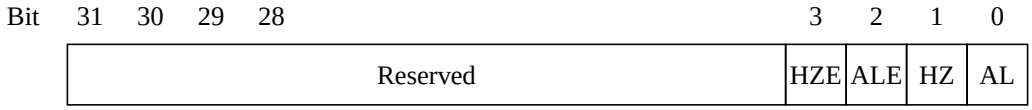


Figure 13. RTC Status Register (RTSR)

The AL (RTC Alarm Detected) and HZ (1 Hz Rising Edge Detected) bits are set if an alarm (i.e. match between the counter and the alarm register) or a rising edge of the 1 Hz clock respectively have been detected.

The ALE (RTC Alarm Interrupt Enable) and HZE (1 Hz Interrupt Enable) bits determine if the corresponding status bits generate a first level interrupt, when set.

The two status bits AL and HZ have to be cleared by writing ones to them to clear the interrupt request.

4.6.1.4 RTC Trim Register (RTTR)

The RTTR is programmed to select the frequency of the “1 HZ” clock. If the register is not programmed but instead left at it’s reset value (all zeros) then the clock will actually run at 32.768 KHz.

The register is divided into two areas, bits 0-15 for the Clock Divider Count, referred to as C[15..0], and bits 16-25 for the Trim Delete Count, referred to as D[9..0].

The trim operation is described in the next section.

4.6.2 RTC Trim Procedure

The 1 Hz clock that is used to increment the RTC counter register is obtained by dividing the output of the 32.768 KHz oscillator. In theory using a 15 bit counter will generate exactly a one Hz clock each time the counter overflows. In reality inherent inaccuracies of the crystals and parasitic capacitances will cause the timebase to be inaccurate. Through the trim procedure the 1 Hz clock can be adjusted to an accuracy of +/- 5 seconds per month.

Following reset the RTTR is initialized to zero, thus disabling the trim mechanism. This results in the 32.768 KHz oscillator output directly feeding the RTC.

To compute the trim value first the actual output frequency of the 32.768 KHz oscillator must be determined. Configuring GPIO pin 27 as an output and selecting the alternate function for this pin makes the clock signal externally available where it can be measured. Trimming the clock is done by dividing the oscillator output by an integer value and then doing fine grain fractional adjustment by periodically deleting clocks from the stream feeding this integer divider:

The integer part of the measured clock is loaded into the C0-C15 field of the RTTR. This value is compared against a 16 bit counter clocked by the 32.768 KHz oscillator. The counter is reset and generates a pulse when the two values are equal. This generates the 1 HZ signal.

For the fine adjustment once every 2^{10}-1 seconds (approximately 17 minutes) a programmable number of 0 to 1023 (2^{10}-1) 32.768 KHz clocks are deleted from the stream feeding the 16 bit counter. This value is set via the D0-D9 field of the RTTR.

The relationship between the oscillator output (f32k) and the generated 1 Hz clock (f1) is given by the following formula:

$$f1 = \frac{(2^{10}-1)\bullet(C[15\ldots0]+1)-D[9\ldots0]}{(2^{10}-1)\bullet(C[15\ldots0]+1)}\bullet\frac{f32k}{(C[15\ldots0])+1}$$

Two examples for trimming taken from [6]:

1. measured oscillator frequency has no fractional component:

   The oscillator output is measured to be 36045.00 Hz. This output is exactly 3277 cycles above the nominal frequency of the crystal and has no fractional component. Therefore only the integer trim function is needed, whereas the D0-D9 field of the RTTR is left at zero to disable the fractional trimming. The integer trim field of RTTR, C0-C15 is programmed with 36045-1 or 0x8CCC. This example leaves an error of zero.

2. measured oscillator frequency has a fractional component:

   The oscillator output is measured to be 32768.92 Hz. Therefore by trimming the clock must be adjusted in a way that on average 32768.92 cycles are counted before a 1 Hz pulse is generated. In analogy to example one the integer field D0-D15 is loaded with 32768-1 or 0x7FFF. As the real clock frequency is 0.92 Hz faster than the integer value the resulting 1 Hz clock will be slightly too fast and must be further slowed down. On average 0.92 cycles per second have to be deleted. As the trimming procedure is performed once every 2^{10}-1 = 1023 seconds, 0,92*1023=941.16 clocks have to be deleted every 1023 seconds. The fractional trimming field D0-D9 therefore has to be loaded with 941 or 0x3AD. The remaining fraction of 0.16 cannot be trimmed out and results in a trimming error of 0.16 cycles per 1023 seconds.

   See chapter 10. for a description on how to do the trimming in an automatic way.

The error calculation yields (in Parts-Per-Million or PPM):

$$\frac{0.16cycles}{1023\,\mathrm{sec}onds}\bullet\frac{1second}{32768cycles} = 0.0048PPM$$

For comparison the maximum error and the guaranteed real time clock accuracy are also calculated:

The maximum error is smaller than 1 clock cycle per 2^{10}-1 seconds, thus the maximum error is:

$$Maximumerror < \frac{1cycle}{1023\,\mathrm{sec}onds}\bullet\frac{1\,\mathrm{sec}ond}{32768cycles} = 0.03PPM$$

To maintain the guaranteed accuracy of +/-5 seconds per month the required accuracy is:

$$\frac{5\,\mathrm{sec}onds}{month}\bullet\frac{1month}{2592000\,\mathrm{sec}onds} = 1.9PPM$$

This indicates that the accuracy provided by the SA-1100 trim mechanism is good enough to compensate for static environmental and manufacturing variables and still provide acceptable accuracy.

Extending the above calculations to a year shows that the maximum accumulated error during one year is less than one second:

$$Maximumerror < \frac{1cycle}{1023\,\mathrm{sec}onds}\bullet\frac{1\,\mathrm{sec}ond}{32768cycles}\bullet\frac{31536000\,\mathrm{sec}onds}{year} = \frac{0.94\,\mathrm{sec}onds}{year}$$

4.6.3 Real Time Clock Declarations

Some macros to use the Real Time Clock unit follow, a sample program can be found in Appendix B.3 ”Real Time Clock Example”.

```
/*
 *      Description:
 *          Type declarations and macros for use with the operating
 *          system timer.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#ifndef util_realtime_h
#define util_realtime_h

#include <util/util_misc.h>
#include <util/util_interrupt.h>

#define REALTIME_BASE   0x90010000

// status register bits
#define REALTIME_AL     0x1
#define REALTIME_HZ     0x2
#define REALTIME_ALE    0x4
#define REALTIME_HZE    0x8


// Real Time Clock Register Offsets
#define RTAR    0x00    // RTC Alarm Register
#define RCNR    0x04    // RTC Count Register
#define RTTR    0x08    // RTC Timer Trim Register
#define RTSR    0x10    // RTC Status Register


// Attention: take care when writing to the status register (RTSR) as AL and
// HZ are status bits which are readeable and cleared by writing ones to them
// while ALE and HZE are normal control bits (thus operations of the type
// reg |= ctrl_bit to set a control bit are not possible as they would clear
// any set status bits as an unwanted and unexpected side-effect)


// get the current count of the Real Time Counter Register
#define REALTIME_GET_COUNT  REG(REALTIME_BASE, RCNR)

// set the value of the Real Time Counter Register
#define REALTIME_SET_COUNT(val) (REG(REALTIME_BASE, RCNR) = (val))

// get the current value of the alarm register
#define REALTIME_GET_ALARM  REG(REALTIME_BASE, RTAR)

// set the alarm register to a new (abs. or relative to current count) value
#define REALTIME_SET_ALARM(val) (REG(REALTIME_BASE, RTAR) = (val))
#define REALTIME_INC_ALARM(val) (REG(REALTIME_BASE, RTAR) = \
                                                (val)+REALTIME_GET_COUNT)


// clear the status bits and disable the interrupts
#define REALTIME_RESET_FLAGS (REG(REALTIME_BASE, RTSR) = \
                                        REALTIME_AL | REALTIME_HZ)
```

Listing 7. Real Time Clock declarations and Macros [file util\util\realtime.h]

```
// get the status of the real time clock - has an alarm or 1 HZ tick occured ?
#define REALTIME_GET_STATUS REG(REALTIME_BASE, RTSR)
#define REALTIME_GET_STATUS_ALARM (REG(REALTIME_BASE, RTSR) & REALTIME_AL)
#define REALTIME_GET_STATUS_ONE_HZ (REG(REALTIME_BASE, RTSR) & REALTIME_HZ)

// enable/disable interrupt on 1 HZ rising edge
// Attention: take care not to clear the AL and HZ bits by or-operations
#define REALTIME_ENABLE_ONE_HZ_INT \
   (REG(REALTIME_BASE, RTSR) = (REG(REALTIME_BASE, RTSR) & 0x0C) | REALTIME_HZE)
#define REALTIME_DISABLE_ONE_HZ_INT (REG(REALTIME_BASE, RTSR) &= 0x4)

// query/clear status of 1 HZ interrupt enable bit
// Attention: take care not to clear the AL bit when clearing the HZ bit by
// just or'ing in REALTIME_HZ !!
#define REALTIME_ONE_HZ_INT_ENABLE_Q (REG(REALTIME_BASE, RTSR) & REALTIME_HZE)
#define REALTIME_CLEAR_ONE_HZ_INT (REG(REALTIME_BASE, RTSR) &= 0xFFFFFFFE)

// enable/disable interrupt both on 1 HZ rising edge AND alarm match
// Attention: take care not to clear the AL and HZ bits by or-operations
#define REALTIME_ENABLE_ONE_HZ_ALARM_INT    \
        (REG(REALTIME_BASE, RTSR) = (REALTIME_HZE | REALTIME_ALE))
#define REALTIME_DISABLE_ONE_HZ_ALARM_INT   \
        (REG(REALTIME_BASE, RTSR) = 0)

// enable/disable interrupt on alarm match
// Attention: take care not to clear the AL and HZ bits by or-operations
#define REALTIME_ENABLE_ALARM_INT \
  (REG(REALTIME_BASE, RTSR) = (REG(REALTIME_BASE, RTSR) & 0x0C) | REALTIME_ALE)
#define REALTIME_DISABLE_ALARM_INT (REG(REALTIME_BASE, RTSR) &= 0x8)

// query/clear status of alarm interrupt enable bit
// Attention: take care not to clear the HZ bit when clearing the AL bit  by
// just or'ing in REALTIME_AL !!
#define REALTIME_ALARM_INT_ENABLE_Q (REG(REALTIME_BASE, RTSR) & REALTIME_ALE)
#define REALTIME_CLEAR_ALARM_INT (REG(REALTIME_BASE, RTSR) &= 0xFFFFFFFD)


// for trimming the clock
#define REALTIME_SET_TRIM(divider_c, delete_c)  \
       (REG(REALTIME_BASE, RTTR) = (delete_c << 16) | divider_c)

#endif //util_realtime_h
```

Listing 7. Real Time Clock declarations and Macros [file util\util\realtime.h]

4.7 Operating System Timers

The SA-1100 contains a 32 bit counter which is clocked by the 3.6864 MHz oscillator and can be used as an operating system timer. This Operating System Count Register (OSCR) is a free running monotonically increasing counter which is not cleared during any reset and thus contains an unknown value after reset. In addition the OS timer unit contains four 32 bit match registers (OSMR[3:0]) which can be read and written. When the value of the OSCR matches the value of one of the match registers the corresponding bit in the OS Timer Status Register (OSSR) is set, if the corresponding bit in the OS Timer Interrupt Enable Register (OIER) is also set. The status bits are also routed to the Interrupt Controller and can be unmasked to cause an interrupt if a match occurs. Match register 3 additionally serves as a watchdog match register which resets the SA-1100 when a match occurs. Except for the Watchdog Match Enable Register (reset to 0) all registers contain an unknown value after reset and have to be initialized by the user before the corresponding FIQ or IRQ interrupts are enabled.

4.7.1 Register Description

Address	Name	Description	Access
0x9000 0000	OSMR[0]	OS Timer Match Register 0	R/W
0x9000 0004	OSMR[1]	OS Timer Match Register 1	R/W
0x9000 0008	OSMR[2]	OS Timer Match Register 2	R/W
0x9000 000C	OSMR[3]	OS Timer Match Register 3	R/W
0x9000 0010	OSCR	OS Timer Counter Register	R/W
0x9000 0014	OSSR	OS Timer Status Register	R/W
0x9000 0018	OWER	OS Timer Watchdog Enable Register	R/W
0x9000 001C	OIER	OS Timer Interrupt Enable Register	R/W

Table 10. OS Timer register block

4.7.1.1 OS Timer Count Register (OSCR)

The OS Timer Count Register is a 32 bit counter which is incremented on rising edges of the 3.6864 MHz clock. The register can be read and written at any time and contains an unknown value after reset.

4.7.1.2 OS Timer Match Registers 0-3 (OSMR[0], OSMR[1], OSMR[2], OSMR[3])

These are 32 bit read- and writable registers which are compared against the OSCR following every rising edge of the 3.6864 MHz clock. If any of these registers matches the counter at this time, the corresponding status bit in the OSSR is set, provided that the corresponding enable bit is also set. OSMR[3] can be used as a watchdog timer.

4.7.1.3 OS Timer Watchdog Enable Register (OWER)

This register contains a single bit at position 0 to enable or disable the watchdog function. The bit is set by writing a one to it and can only be cleared by one of the reset functions (hard reset, software reset, watchdog reset) or by entering sleep mode. If the bit is zero then OS timer match register[3] matches cause a normal OS Timer match interrupt request, if the bit is set to one, match register[3] matches cause a reset of the SA-1100.

4.7.1.4 OS Timer Status Register (OSSR)

The status register contains four bits (bits 0-3) indicating whether a match has occurred on any of the four match registers since the last clear. The bits are set when a match occurs (following the rising edge of the 3.6864 MHz clock) and must be cleared by writing a one to the proper bit position. Writing zeros to this register has no effect.

4.7.1.5 OS Timer Interrupt Enable Register (OIER)

This register contains four enable bits (bits 0-3) that determine whether a match between one of the match registers and the count register will set the corresponding status bit in the OSSR. If a status bit is already set, then clearing the corresponding interrupt enable bit will not clear the status bit. If both the corresponding OIER and OSSR bits are set a level 1 interrupt request occurs.

4.7.2 Watchdog Timer

OS Timer Match Register[3] can also be used as a watchdog compare register. If a match between this register and the count register is detected and the watchdog has been enabled (by setting bit 0 in the OWER), reset is applied to the SA-1100. The internal reset is activated for 256 processor clocks and then removed, allowing the SA-1100 to reboot. The Power Manager, Refresh Timer, and the PLL configuration do not receive this reset. Following a watchdog reset a status bit is set in the Reset Controller Status Register (RCSR) (see [6].)

To use OSMR[3] as a watchdog the OS should periodically read the count register (OSCR), add a number to the value read and write the resulting value to OSMR[3]. The added value corresponds to the amount of time before the next time-out. If the OS fails to repeat this procedure within the programmed time, i.e. before the match occurs, the match will cause a watchdog reset, thus forcing the execution of the reset code.

4.7.3 OS Timer Declarations and Functions

The following macros can be used to directly access the various OS Timer registers:

```
/*
 *      Description:
 *          Type declarations and macros for use with the operating
 *          system timers.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#ifndef util_ostimer_h
#define util_ostimer_h

#include <util/util_misc.h>
#include <util/util_interrupt.h>

#define OSTIMER_BASE    0x90000000

#define OSTIMER_CHANNEL_0   0
#define OSTIMER_CHANNEL_1   1
#define OSTIMER_CHANNEL_2   2
#define OSTIMER_CHANNEL_3   3

/* OS Timer Register Offsets */
#define OSMR0   0x00    /* OS Timer Match Register 0 */
#define OSMR1   0x04    /* OS Timer Match Register 1 */
#define OSMR2   0x08    /* OS Timer Match Register 2 */
#define OSMR3   0x0C    /* OS Timer Match Register 3 */
#define OSCR    0x10    /* OS Timer Counter Register */
#define OSSR    0x14    /* OS Timer Status Register */
#define OWER    0x18    /* OS Timer Watchdog Enable Register */
#define OIER    0x1C    /* OS Timer Interrupt Enable Register */

/* status register bits */
#define OSSR_M0 BIT0
#define OSSR_M1 BIT1
#define OSSR_M2 BIT2
#define OSSR_M3 BIT3

/* interrupt enable register bits */
#define OIER_E0 BIT0
#define OIER_E1 BIT1
#define OIER_E2 BIT2
#define OIER_E3 BIT3

/* get/set the current value of the OS Timer Counter Register */
#define OSTIMER_GET_COUNT REG(OSTIMER_BASE,OSCR)
#define OSTIMER_SET_COUNT(n) (REG(OSTIMER_BASE, OSCR)=(n))



/* Note: the following macros with parameter “channel” are intended
 *       to be used with the previously defined constants
```

Listing 8. OS Timer declarations and macros [file util\util_ostimer.h]

```
 *       OSTIMER_CHANNEL_0 - OSTIMER_CHANNEL_3.
 *       Using the constants OSSR_M0..OSSR_M3 and OIER_E0..OIER_E3 and
 *       redefinition of the macros would allow you to manipulate more than
 *       one channel a time, e.g.:
 *       #define OSTIMER_ENABLE_INT(arg) REG(OSTIMER_BASE, OIER) |= (arg)
 *
 *       OSTIMER_ENABLE_INT(OIER_E0 | OIER_E2);
 *
 *       I chose the other method however to allow easy manipulation in
 *       combination with the interrupt channels (see functions in
 *       util_ostimer.c)
 */

/* set the specified channel (0 to 3) to the given (absolute) value */
#define OSTIMER_SET_MATCH_REG(channel, val) \
              (REG(OSTIMER_BASE,((channel)*4)) = (val))

/* set the specified channel (0 to 3) to the current
 * counter value+given increment
 * Attention: before unmasking the corresponding interrupt the appropriate
 * status bit should be reset to make sure it is not set from an earlier
 * match. This step can be omitted if one knows that the status bit has
 * been reset a reasonably short time ago (e.g., in an interrupt handler that
 * resets the status bit and shortly afterwards sets the new timeout)
 */
#define OSTIMER_INC_MATCH_REG(channel, inc) \
 (REG(OSTIMER_BASE,((channel)*4))=REG(OSTIMER_BASE,OSCR)+(inc))

/* enable interrupt for the given channel (0 to 3) */
#define OSTIMER_ENABLE_INT(channel) \
   (REG(OSTIMER_BASE,OIER) |= 1<<(channel))

/* disable interrupt for the given channel (0 to 3) */
#define OSTIMER_DISABLE_INT(channel) \
   (REG(OSTIMER_BASE,OIER) &= ~(1<<(channel)))

/* disable interrupts for all channels and reset match flags */
#define OSTIMER_DISABLE_INT_ALL  REG(OSTIMER_BASE,OIER) = 0; \
                                 REG(OSTIMER_BASE,OSSR) = 0xf

/* query the match bit for the given channel */
#define OSTIMER_QUERY_MATCH(channel) \
   (REG(OSTIMER_BASE,OSSR) & 1<<(channel))

/* reset status bit for a given channel -> write a one to the
   specified bit, zeros don't matter */
#define OSTIMER_RESET_INT(channel) \
   (REG(OSTIMER_BASE,OSSR) = 1<<(channel))
```

Listing 8. OS Timer declarations and macros [file util\util_ostimer.h]

Some functions that proved useful in combination with the OS Timer functionality:

```
/*
 *      Description:
 *          Various functions for use with the OS timers
 *          First some helper functions for use with the hardware timers,
 *          then a software timer implementation, especially for use with
 *          the IrDA stack.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */
```

Listing 9. Functions to use the OS Timer functionality [file util_ostimer.c]

```
#include <util/util_ostimer.h>
#include <util/util_debug.h>


// enable/disable debug messages
//#define DEBUG_OSTIMER
#ifdef DEBUG_OSTIMER
    extern int debug_initialized;
    #define DEBUG_STRING(c) debug_String(c)
    #define DEBUG_PUTBYTEDIRECT(c) debug_PutByteDirect(c)
#else
    #define DEBUG_STRING(c)
    #define DEBUG_PUTBYTEDIRECT(c)
#endif


// mask and unmask the software timer interrupt
#define OSTIMER_LIST_INT_ON  INT_UNMASK(INT_OSTIMER_0+ostimer_list_channel)
#define OSTIMER_LIST_INT_OFF INT_MASK(INT_OSTIMER_0+ostimer_list_channel)


// keep info about the four timer channels (handler and if a handler
// has gone off (used in the generic timer handler ostimer_TimerHandler)
volatile ostimer_timer_info ostimer_info[4] = { {NULL, FALSE}, {NULL, FALSE},
                                                {NULL, FALSE}, {NULL, FALSE} };

/*
 * Function: ostimer_TimerHandler
 *  Purpose: Default OS Timer handler
 *
 * Parameters: Interrupt handler arguments passed by Angel.
 *    Returns: void
 *
 * If the timer goes off, the interrupt request is reset and
 * ostimer_info[channel].gone_off is set to true.
 */
void ostimer_TimerHandler(unsigned int ident, unsigned int data, unsigned int
empty_stack)
{
    int channel = ident-INT_OSTIMER_0;
    OSTIMER_RESET_INT(channel);
    ostimer_info[channel].gone_off = TRUE;
}


/*
 * Function: ostimer_InitChannelDefault
 *  Purpose: Install a default OS timer handler.
 *
 * Parameters:
 *      Input: channel            the channel for which the handler is to
 *                                be installed
 *             update             install handler in any case or only if no
 *                                handler has been installed yet.
 *    Returns: 0                  The new handler has been installed
 *             -1                 There was already a handler installed
 *
 * Install a standard timer for the given channel.
 * If update=TRUE install in any case, if update=FALSE install the
 * timer only if no timer has been installed on this channel yet.
 */
int ostimer_InitChannelDefault(int channel, BOOL update)
{
```

Listing 9. Functions to use the OS Timer functionality [file util_ostimer.c]

```
    if(ostimer_info[channel].handler && !update)
    {
        return -1;
    }
    else
    {
        OSTIMER_ENABLE_INT(channel);
        int_InstallHandler(INT_OSTIMER_0+channel, ostimer_TimerHandler);
        ostimer_info[channel].handler = ostimer_TimerHandler;
        return 0;
    }
}


/*
 * Function: ostimer_InitChannelCustom
 *  Purpose: Install a custom OS timer handler.
 *
 * Parameters:
 *      Input: channel           the channel for which the handler is to
 *                               be installed
 *             handler           the new handler to be installed
 *             update            install handler in any case or only if no
 *                               handler has been installed yet.
 *    Returns: 0                 The new handler has been installed
 *             -1                There was already a handler installed
 *
 * Install a custom timer handler for the given channel.
 * If update=TRUE install in any case, if update=FALSE install the
 * timer only if no timer has been installed on this channel yet.
 */
int ostimer_InitChannelCustom(int channel, angel_IntHandlerFn handler, BOOL update)
{
    if(ostimer_info[channel].handler && !update)
    {
        return -1;
    }
    else
    {
        OSTIMER_ENABLE_INT(channel);
        int_InstallHandler(INT_OSTIMER_0+channel, handler);
        ostimer_info[channel].handler = handler;
        return 0;
    }
}


/*
 * Function: ostimer_RemoveChannel
 *  Purpose: Remove a timer from the given channel.
 *
 * Parameters:
 *      Input: channel           the channel which is to be reset and disabled.
 *
 *    Returns: 0                 The channel has been reset and disabled.
 *             -1                The channel was not used
 *
 * This function disables the interrupt settings and resets the
 * appropriate ostimer_info entry.
 */
int ostimer_RemoveChannel(int channel)
{
    if(ostimer_info[channel].handler)
    {
        OSTIMER_DISABLE_INT(channel);
```

Listing 9. Functions to use the OS Timer functionality [file util_ostimer.c]

```
        INT_MASK(INT_OSTIMER_0+channel);
        ostimer_info[channel].handler = NULL;
        ostimer_info[channel].gone_off = FALSE;
        return 0;
    }
    else
    {
        return -1;
    }
}


/*
 * Function: ostimer_WaitTime
 *  Purpose: Wait for a certain time or condition in a busy loop
 *
 * Parameters:
 *      Input: channel            the channel to be used for the wait
 *             time               the amount of time to wait
 *             unit               the unit ('u' = us, 'm' = ms, 's' = sec.) to
 *                                wait ('s' is default, i.e., every value
 *                                except 'u' and 'm' is interpreted as seconds)
 *             *cond              a condition which stops the waiting loop when
 *                                set to true
 *    Returns: TRUE               The given timeout has passed.
 *             FALSE              *cond was set to true before the timeout was
 *                                reached
 *
 * Wait on the given channel for either the given time or until
 * cond becomes TRUE. The function does a busy wait until either
 * of the conditions is met.
 * The timer has to be installed before calling the function.
 * Return is TRUE if the timer went off, FALSE if *cond became true
 * before the timer went off.
 */
BOOL ostimer_WaitTime(int channel, int time, char unit, BOOL *cond)
{
    int inc;
    switch(unit)
    {
        case 'u':   inc = 3.6864*time;
                    break;
        case 'm':   inc = 3686.4*time;
                    break;
        default:    inc = 3686400*time;
    }
    ostimer_info[channel].gone_off = FALSE;

    OSTIMER_INC_MATCH_REG(channel, inc);
    OSTIMER_RESET_INT(channel);

    INT_UNMASK(INT_OSTIMER_0+channel);
    while(!(ostimer_info[channel].gone_off || *cond) )
        ;
    return ostimer_info[channel].gone_off;
}


/*
 * Function: ostimer_WaitCount
 *  Purpose: Wait for a certain number of clock ticks or condition in a busy loop
 *
 * Parameters:
 *      Input: channel            the channel to be used for the wait
 *             inc                the number of (3.6864MHz) clock ticks to wait
```

Listing 9. Functions to use the OS Timer functionality [file util_ostimer.c]

```
 *             *cond             a condition which stops the waiting loop when
 *                               set to true
 *    Returns: TRUE              The given timeout has passed.
 *             FALSE             *cond was set to true before the timeout was
 *                               reached
 *
 * Wait on the given channel for either inc ticks of the system timer
 * clock or until *cond becomes TRUE. The function does a busy wait
 * until either of the conditions is met.
 * The timer has to be installed before calling the function.
 * Return is TRUE if the timer went off, FALSE if *cond became true
 * before the timer went off.
 */
BOOL ostimer_WaitCount(int channel, int inc, BOOL *cond)
{
    ostimer_info[channel].gone_off = FALSE;
    OSTIMER_INC_MATCH_REG(channel, inc);
    OSTIMER_RESET_INT(channel);
    INT_UNMASK(INT_OSTIMER_0+channel);
    while(!(ostimer_info[channel].gone_off || *cond))
        ;
    return ostimer_info[channel].gone_off;
}
```

Listing 9. Functions to use the OS Timer functionality [file util_ostimer.c]

4.7.4 Software Timer Implementation

Implementation of the IrDA protocol stack required a number of different timers, therefore I implemented “software timers”, similar to the Linux timers. For each timer a structure of type `timer_list` is created and filled in with values for the desired timeout (field `expires`), a function pointer for a callback function to be executed when the timer expires (field `function`) and an optional value to be passed to the callback function (field `data`). This structure is passed to the `ostimer_AddTimer` function. The software timers operate based on the global variable `ostimer_ticks`.

The function `ostimer_InitListInt()` initializes the necessary data structures and installs an interrupt handler for the supplied OS Timer channel. This handler resets the interrupt status bit, sets the new value for the match register and increments the variable `ostimer_ticks`. Then it calls `Angel_SerialiseTask` (see section 4.2) to serialize the function `ostimer_CheckTimers` to check the active software timers for expiry, executing in supervisor mode, but with interrupts enabled. In the current implementation the timer resolution is 1ms, but can easily be changed via the constant `OSTIMER_LIST_TIMER_INC` which is the increment for the match register. If the timeout values for the timers are given in multiples/fractions of seconds by using the constant SEC (e.g. `20*SEC/1000` for 20ms), then the timeouts are not affected by changes in the resolution.

The serialized function `ostimer_CheckTimers` compares the `expires`-fields of all currently registered timers with the value in `ostimer_ticks`. If it detects a match it removes the timer from the timer queue and queues the corresponding callback function for later execution in user mode via a call to `Angel_QueueCallback` (see section 4.2). Finally it checks the variable `ostimer_bh_marked` and if it is not zero, it queues the function `ostimer_QueueBH`. `ostimer_bh_marked` can be set via calling `ostimer_MarkBH(flag)` with an appropriate value for `flag` to request a certain service. This is similar to the Linux bottom half mechanisms and allows to regularly check for certain conditions and execute appropriate functions if one of the conditions is true. Currently three conditions are implemented to support receiving and transmitting within the IrDA protocol stack. For more details see Chapter 9. ”IrDA Implementation”.

Declarations for the software timer functionality:

```
#define OSTIMER_CLOCKRATE 3686400
#define OSTIMER_LIST_TIMER_INC 3686  // about every ms
//#define OSTIMER_LIST_TIMER_INC 2510  // seems to be minimally possible value
```

Listing 10. Declarations for the software timer functionality [file util\util_ostimer.h]

```
// one sec, used to specify time outs independant of the timer resolution
#define SEC OSTIMER_CLOCKRATE/OSTIMER_LIST_TIMER_INC

typedef void (*TIMER_CALLBACK)(unsigned long);


// software timer structure
struct timer_list {
    struct timer_list *next; /* MUST be first element */
    struct timer_list *prev;
    unsigned long expires;
    unsigned long data;
    TIMER_CALLBACK function;
//    void (*function)(unsigned long);
};



/*
 * Function: ostimer_InitTimer
 *  Purpose: Init the supplied software timer structure
 *
 * Parameters:
 *      Input: timer              the timer structure to be initialized.
 *
 *    Returns: void
 *
 * This function should be called for every software timer before it is
 * passed to ostimer_AddTimer.
 */
__inline void ostimer_InitTimer(struct timer_list * timer)
{
    timer->next = NULL;
    timer->prev = NULL;
}


#ifdef IRDA

// extern function prototypes
void irport_BHTransmit(void);
void irport_BHReceive(void);


// used to set bottom half handler service requests
extern volatile int ostimer_bh_marked;

// flags to request bottom half service
#define OSTIMER_BH_IRPORT_TRANSMIT  0x1
#define OSTIMER_BH_IRPORT_RECEIVE   0x2
#define OSTIMER_BH_IRPORT_UNWRAP    0x4


/*
 * Function: ostimer_MarkBH
 *  Purpose: set a bottom half handler service request
 *
 * Parameters:
 *      Input: mode               the service to be requested, a combination of
 *                                the above defined flags
 *    Returns: void
 */
__inline void ostimer_MarkBH(int mode)
{
```

Listing 10. Declarations for the software timer functionality [file util\util_ostimer.h]

```
    ostimer_bh_marked |= (mode);
}


/*
 * Function: ostimer_UnmMarkBH
 *  Purpose: clear a bottom half handler service request after execution
 *
 * Parameters:
 *      Input: mode              the service request to be cleared,
 *                               a combination of the above defined flags
 *    Returns: void
 */
__inline void ostimer_UnmarkBH(int mode)
{
    ostimer_bh_marked &= ~mode;
}


/*
 * Function: ostimer_IsMarkedBH
 *  Purpose: check if any bottom half handler service is requested
 *
 * Parameters: none
 *    Returns: 0                 no bh service requested
 *             other             bh service requested
 */
__inline bool ostimer_IsMarkedBH(void)
{
    return ostimer_bh_marked;
}

#endif // IRDA
```

Listing 10. Declarations for the software timer functionality [file util\util_ostimer.h]

Code that implements the software timer functionality (including the bottom half handler mechanisms used in the IrDA implementation):

```
/* to check if the Angel function hooks have been initialized */
extern int misc_initialized;

/* the interrupt handler for the software timer interrupt */
void ostimer_ListHandler(unsigned int ident, unsigned int data, unsigned int
empty_stack);

/* the list of currently active software timers */
static struct timer_list ostimer_timer_list;

/* the OS timer channel to be used for the software timers */
volatile static int ostimer_list_channel;

/* the current software timer "time count" */
volatile unsigned long ostimer_ticks = 0;




// block all ostimer tasks during SA1100 SIR (sofware modulation)
// transmission
volatile int ostimer_blocked = FALSE;


#ifdef IRDA
```

Listing 11. Functions for SW timers and bottom half handling

```
// global variables needed for IrDA bottom half handling

// which bottom half handler is to execute
volatile int ostimer_bh_marked = 0;

// if true, then check for bh tasks in the timer interrupt
volatile int ostimer_bh_active = 0;

// prevent multiple queuing of the ostimer bh function
volatile int ostimer_bh_pending = FALSE;

#endif  // IRDA


/*
 * Function: ostimer_InitListInt
 *  Purpose: Init the software timer functionality
 *
 * Parameters:
 *      Input: channel            the OS timer channel to be used for the
 *                                software timers
 *
 *    Returns: 0                  the software timer functionality was
 *                                properly initialized
 *             -1                 the supplied timer channel was already
 *                                used by another handler
 *
 * Initialize the software timer data structures and install the interrupt
 * handler. The interrupt remains masked until the first timer is set.
 * If a handler is already set for the given channel (ostimer_info[])
 * the function returns -1, otherwise 0 to signal success.
 * If DEBUG_OSTIMER is defined, the function checks if debug_Init()
 * has been called and calls it if necessary. Further it executes
 * misc_InitAngelFunctions if necessary.
 */
int ostimer_InitListInt(int channel)
{

#ifdef DEBUG_OSTIMER
    if(!debug_initialized)
    {
        debug_Init();
    }
#endif

    if(!misc_initialized)
    {
        misc_InitAngelFunctions();
    }

    // init the timer list
    ostimer_timer_list.next = NULL;
    ostimer_timer_list.prev = &ostimer_timer_list;

    if(ostimer_info[channel].handler)
    {
        return -1;
    }
    // set up the interrupt
    OSTIMER_ENABLE_INT(channel);
    int_InstallHandler(INT_OSTIMER_0+channel, ostimer_ListHandler);

    // update the global variable ostimer_list_channel
    ostimer_list_channel = channel;
    ostimer_info[channel].handler = ostimer_ListHandler;
```

Listing 11. Functions for SW timers and bottom half handling

```
    return 0;
}


/*
 * Function: ostimer_AddTimer
 *  Purpose: Add a software timer
 *
 * Parameters:
 *      Input: timer             structure describing the timer to add
 *
 *    Returns: void
 *
 * When the first timer is added to the list of software timers the timer
 * interrupt is enabled. The supplied timer is added to the timer list.
 * To guarantee atomic access to the timer list the function is guarded
 * by Angel_EnterSVC()/Angel_ExitToUSR().
 */
void ostimer_AddTimer(struct timer_list * timer)
{
    // switch to SVC mode
    Angel_EnterSVC();

    // activate the interrupt when first timer is added to list
    if(ostimer_timer_list.next == NULL)
    {
        OSTIMER_INC_MATCH_REG(ostimer_list_channel, OSTIMER_LIST_TIMER_INC);
        INT_UNMASK(INT_OSTIMER_0+ostimer_list_channel);
    }

    // append new timer to the list
    timer->prev = ostimer_timer_list.prev;
    (ostimer_timer_list.prev)->next = timer;
    ostimer_timer_list.prev = timer;
    Angel_ExitToUSR();
}


/*
 * Function: ostimer_DelTimer
 *  Purpose: Remove a software timer
 *
 * Parameters:
 *      Input: timer             structure describing the timer to remove
 *
 *    Returns: 0                 the timer could be removed
 *             -1                the supplied timer was not found in the list
 *                               of currently active software timers.
 *
 * To guarantee atomic access to the timer list first supervisor mode is
 * entered. Then the supplied timer is searched and removed from the list, if
 * found. Finally if there are no timers left and bottom half handler services
 * are not active the timer interrupt is disabled.
 */
int  ostimer_DelTimer(struct timer_list * timer)
{
    struct timer_list* temp;
    int on = TRUE;


    // disable timer interrupt to prevent simultaneous manipulation
    // of the list
    Angel_EnterSVC();

    temp = ostimer_timer_list.next;
```

Listing 11. Functions for SW timers and bottom half handling

```
    // find given timer in list
    while(temp != timer && temp != NULL)
    {
        temp = temp->next;
    }

    if(temp != NULL)
    {
        // if timer was found, remove it

        (temp->prev)->next = temp->next;

        // remove timer from end of list
        if(temp->next == NULL)
        {
            ostimer_timer_list.prev = temp->prev;

            // if timer was last timer in list stop the interrupt
#ifdef IRDA
            if((ostimer_timer_list.prev == &ostimer_timer_list) &&
                !ostimer_bh_active)
#else
            if(ostimer_timer_list.prev == &ostimer_timer_list )
#endif
            {
                DEBUG_STRING("timer int shut off\n");
                on = FALSE;
            }
        }
        // remove timer from within the list
        else
        {
            (temp->next)->prev = temp->prev;
        }
        temp->next = NULL;
        temp->prev = NULL;

        // if there are timers left or bh services requested, then reenable the
        // interrupt.
        if(!on)
            OSTIMER_LIST_INT_OFF;

        Angel_ExitToUSR();
        return 0;
    }
    Angel_ExitToUSR();
    return -1;

}


#ifdef IRDA


/*
 * Function: ostimer_SetBHActive
 *  Purpose: enable bottom half handler services
 *
 * Parameters: none
 *    Returns: void
 */
void ostimer_SetBHActive(void)
{
    ostimer_bh_active = TRUE;
```

Listing 11. Functions for SW timers and bottom half handling

```
}


/*
 * Function: ostimer_SetBHInactive
 *  Purpose: disable bottom half handler services
 *
 * Parameters: none
 *    Returns: void
 */
void ostimer_SetBHInactive(void)
{
    ostimer_bh_active = FALSE;
}


/*
 * Function: ostimer_QueueBH
 *  Purpose: call a bottom half handler function
 *
 * Parameters:
 *      Input: arg0, arg1,
 *             arg2, arg3         not used
 *
 *    Returns: void
 *
 * If called at least one bottom half handler flag is marked, therefore
 * test for the flags and execute the appropriate function(s)
 */
static void ostimer_QueueBH(void * arg0, void* arg1, void* arg2, void* arg3)
{

    ostimer_bh_pending = FALSE;

    // transmit service requested
    if(ostimer_bh_marked & OSTIMER_BH_IRPORT_TRANSMIT)
    {
        irport_BHTransmit();
    }

    // receive service requested (i.e. complete packet ready to be passed
    // up the protocol stack
    if(ostimer_bh_marked & OSTIMER_BH_IRPORT_RECEIVE)
    {
        irport_BHReceive();
    }

    // received bytes to be unwrapped
    if(ostimer_bh_marked & OSTIMER_BH_IRPORT_UNWRAP)
    {
        irport_BHUnwrap();
    }

    if(ostimer_bh_marked & OSTIMER_BH_IRLAN_TRANSMIT)
    {
        // not implemented because not supported by used IP stack
    }


    // invalid flag set
    if(ostimer_bh_marked & ~(OSTIMER_BH_IRPORT_TRANSMIT | OSTIMER_BH_IRPORT_RECEIVE |
        OSTIMER_BH_IRPORT_UNWRAP | OSTIMER_BH_IRLAN_TRANSMIT))
    {
        ostimer_bh_marked = 0;
    }
```

Listing 11. Functions for SW timers and bottom half handling

```
}

#endif // IRDA


/*
 * Function: ostimer_QueueTimerFunction
 *  Purpose: call the passed timer callback function
 *
 * Parameters:
 *      Input: func               the callback function to be executed
 *             timer_data         32 bit argument to be passed to the callback
 *             arg2, arg3         not used
 *
 *    Returns: void
 *
 * This function is just a wrapper because Angel_QueueCallback expects four
 * parameters, while the timer callback functions just use one parameter.
 */
static void ostimer_QueueTimerFunction(TIMER_CALLBACK func, void* timer_data,
                                   void* arg2, void* arg3)
{
    (*func)((unsigned long)timer_data);
}


/*
 * Function: ostimer_CheckTimers
 *  Purpose: Check for expired software timers and bottom half
 *           service requests
 *
 * Parameters:
 *      Input: arg                not used
 *
 *    Returns: void
 *
 * Executed after the call to Angel_SerialiseTask. Checks if any of the
 * current timers has gone off and if so removes it and queues its
 * callback function for execution in user mode.
 * After that it queues the ostimer_QueueBH function if at least one of
 * the bottom half handler flags is set.
 * As a serialized function it has the lock and executes in supervisor
 * mode, therefor no need for guarding Angel_EnterSVC()/Angel_ExitToUSR()
 * calls. If a timer interrupt should occur already during execution of
 * this function, the interrupt handler is executed but the newly queued
 * call to ostimer_CheckTimers() can only be executed after this call
 * has returned, therefor no problem with the list.
 */
static void ostimer_CheckTimers(void* arg)
{
    struct timer_list* temp;
    temp = ostimer_timer_list.next;

    // check timers in the list
    while(temp != NULL)
    {
        if(temp->expires <= ostimer_ticks)
        {

            // remove the expired timer
            (temp->prev)->next = temp->next;

            // remove timer from end of list
            if(temp->next == NULL)
            {
```

Listing 11. Functions for SW timers and bottom half handling

```
                ostimer_timer_list.prev = temp->prev;
            }
            // remove timer from within the list
            else
            {
                (temp->next)->prev = temp->prev;
            }
            temp->next = NULL;
            temp->prev = NULL;


            // queue expired timers for later execution in user mode
            Angel_QueueCallback((angel_CallbackFn)ostimer_QueueTimerFunction,
                TP_AngelCallBack, (void*)(temp->function),(void*)(temp->data),
                NULL,NULL);
        }
        temp = temp->next;
    }

#ifdef IRDA
    // check for bh service requests
    if(ostimer_bh_marked &&!ostimer_bh_pending )
    {
        ostimer_bh_pending = TRUE;

        // queue bh handler for execution in user mode
        Angel_QueueCallback((angel_CallbackFn)ostimer_QueueBH, TP_AngelCallBack,
                NULL, NULL, NULL, NULL);
    }
#endif // IRDA


}


/*
 * Function: ostimer_ListHandler
 *  Purpose: The core software timer interrupt handler
 *
 * Parameters: interrupt handler arguments passed by Angel
 *    Returns: void
 *
 * Just reset the interrupt request flag, set the match register for the
 * next timeout, increment the time base and call Angel_SerialiseTask to do
 * the actual processing, if not blocked.
 */
void ostimer_ListHandler(unsigned int ident, unsigned int data,
                         unsigned int empty_stack)
{

    OSTIMER_RESET_INT(ident-INT_OSTIMER_0);
    OSTIMER_INC_MATCH_REG(ostimer_list_channel, OSTIMER_LIST_TIMER_INC);
    ostimer_ticks++;

    // try to
    if(!ostimer_blocked)
    {
        Angel_SerialiseTask(0, ostimer_CheckTimers, NULL, empty_stack);
    }

}
```

Listing 11. Functions for SW timers and bottom half handling

4.8 UARTs

Three of the five serial ports of the SA-1100 support UART operation. Serial port 3 is a pure UART port and is used with the ARM debugger when using Angel. Serial port 1 offers the SDLC protocol in addition to the UART mode while serial port 2 is intended as an infrared port. It contains a UART with modulation to support slow infrared mode (SIR) as well as an HSSP for highspeed infrared communication (FIR). This section describes how to program the UARTs of the SA-1100 (identical for all three UART units). It also shows how to set up serial port 1 for UART communication. For the SDLC mode see [6]. The details related to serial port 2 infrared communication are covered in Chapter 5. ”Infrared Communication”.

The SA-1100 UARTs are general purpose, full-duplex, universal asynchronous receiver/transmitter units which support most of the functionality of the National Semiconductor 16550 protocol. They can operate at baud rates between 56.24 bps and 230.4 Kbps; support 7 or 8 data bits; odd, even or no parity; one start bit; one or two stop bits; and can transmit a continuos break signal. The external pins dedicated to these units are TXDx and RXDx (x = 1,2,3). The transmit and receive engines can be en-/disabled separately. If an engine is disabled its pin is controlled by the PPC and can be used for general purpose I/O.

Each of the UARTs contains an 8-entry x 8-bit transmit FIFO to buffer outgoing data and a 12-entry x 11-bit receive FIFO to buffer incoming data. The additional three bits in the receiver FIFO are used to store framing, parity and receive FIFO overrun error flags for each character received and stored in the FIFO. The FIFOs can be filled/emptied using DMA or register writes/reads (either polled or interrupt driven). Interrupts are generated when an error (framing, parity or receiver overrun) is present within the bottom four entries of the receive FIFO, when the receive FIFO is one- to two-thirds full, when the receive FIFO is not empty and the receiver has been idle for at least three frame periods, when the transmit FIFO is half-empty and when a begin or end of break is detected by the receiver.

The UART units do not provide modem control signals (RTS, CTS, DTR and DSR), if required these can be implemented using GPIO pins.

After a hardware reset the UARTs are disabled and control of their pins is given to the PPC which configures all pins as inputs. Reset also causes the FIFOs to be flushed. Before enabling a UART all status bits must first be cleared by writing a one to each bit. Then the control registers can be programmed to select the desired mode. If desired the transmit FIFO can be filled with data at this point. Then transmitter and/or receiver can be enabled to start the communication.

Figure 14 shows the UART data frame format.

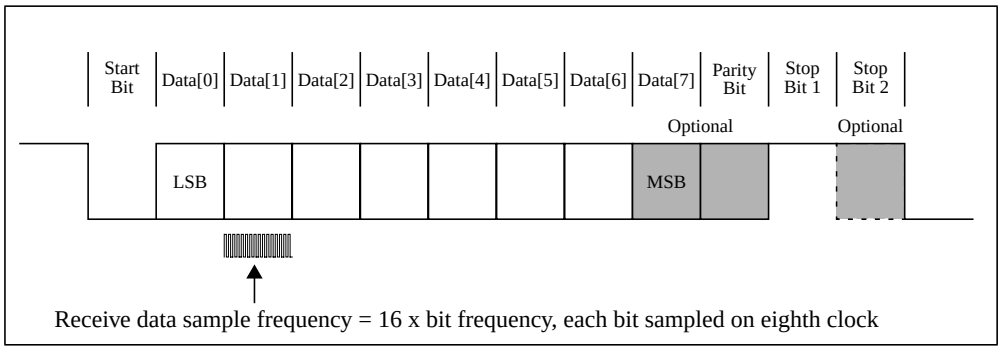


Figure 14. UART Data Frame

Each frame begins with a start bit, represented by a high to low transition. Then, depending on the selected mode, seven or eight data bits are transmitted, starting with the least significant bit. Optionally a parity bit can follow (set if even parity is enabled and the data byte contains an odd number of ones or if odd parity is selected and the data byte contains an even number of ones). The frame ends with one or two stop bits, represented by one or two successive bit periods of logic one. The receiver tests for only one stop bit per frame.

The UARTs use NRZ encoding to represent individual bit values, i.e. logical one is represented by a line transition, while logical zero is represented by no line transition. Figure 15 shows an example of NRZ encoding for the data byte 01001011b:

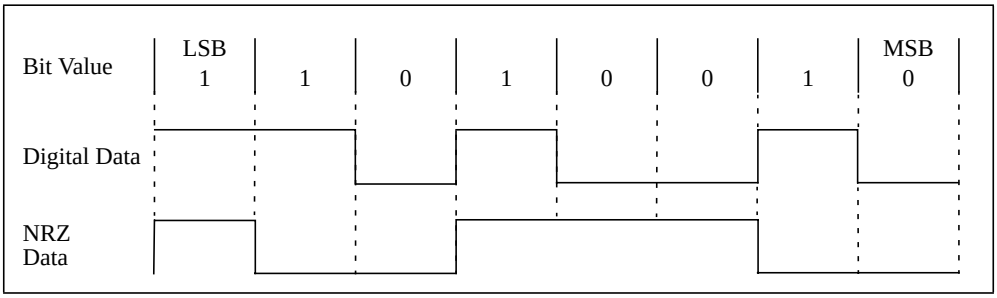


Figure 15. Example for NRZ encoding

The baud rate is generated by dividing down the 3.6864 MHz clock first by a programmable number between 1 and 4097 and then by a fixed value of 16. The receiver clock is synchronized with the incoming data stream using a digital PLL each time the start bit is detected. The bits are then sampled at the middle of each bit period by counting 8 of the 16 clocks which are produced before the fixed divide by 16 takes place.

4.8.1 Receive Operation

The UART receives incoming data using a serial shifter, removes start, parity, and stop bits; then transfers the data into the receive FIFO. If parity is enabled the number of one bits within the frame is counted and checked against the received parity bit. If a parity error is detected the parity error flag in the FIFO is set for the corresponding entry. If a logic zero is detected where a stop bit (logic one) was expected, the corresponding framing error flag is set.

When the FIFO is one- to two-thirds (i.e., five to eight bytes) full, an interrupt or DMA request is signalled. If the FIFO is completely filled and more data are received, the overrun error flag is set for the last valid entry in the FIFO. Any more data received while the FIFO is full is discarded.

The error flags are transferred down the FIFO along with the data that caused the error. Whenever one of the bottom four FIFO entries contains one or more set error bits an interrupt request is generated and receive FIFO DMA requests are disabled until the entry containing the error is flushed out of the FIFO. Each time an entry is transferred to the bottom position of the FIFO the three error flags are transferred to their respective flags in the status register. If set they indicate that the next byte to be read from the FIFO contains an error. Therefore if the “error interrupt” was generated, the flags must be checked and programmed I/O must be used to remove the FIFO entries one byte at a time until the four bottom entries in the FIFO are free of errors. At this point DMA is automatically reenabled.

4.8.2 Transmit Operation

As the UARTs are full-duplex the transmission engine can operate at the same time as the receive engine. Bytes are read from the transmit FIFO, start, optionally parity, and stop bit(s) are added to generate a valid frame which is then loaded into the serial shifter. Its contents are then transferred to the corresponding TXD pin, clocked by the programmed baud clock. When the transmit FIFO is half empty an interrupt or DMA request is generated. When the transmit FIFO is completely emptied, the TXD pin is set to high (one) and remains at this level until additional data is written to the FIFO.

Note: this means that although the transmitter is not busy it powers the TxD pin and the connected line and therefore adds to the power consumption, unless the transmit engine is disabled.

4.8.3 FIFOs

To reduce chip size and power consumption, the UART’s FIFOs use self-timed logic (i.e. they are not clocked). Due to process and environmental variations the depth at which a service request is triggered to empty the receive FIFO is variable. This variation spans a maximum of four FIFO entries. The receive FIFO service request occurs when the FIFO contains between five and eight (of a maximum of twelve) entries.

This variation applies only to the receive FIFO, the transmit FIFO always generates the service request as soon as it contains four or more empty entries and negates the request when the FIFO contains five or more data bytes.

If DMA is used to service either of the FIFOs the burst size must be set to four words, even though more than four valid entries may exist within the receive FIFO. If programmed I/O is used to service the FIFOs four bytes may be written to the transmit FIFO / read from the receive FIFO without any checking. After that the status bits (transmitter not full, receiver not empty) must be checked before writing more bytes to the transmitter or reading more bytes from the receiver FIFO.

4.8.4 Register Description

The UARTs contain seven byte-wide registers, four control registers (UART2 contains one additional control register), one data register and two status registers. The control registers are used to program baud rate, number of stop bits, parity, enable/disable transmit and receive operation, interrupts, and some more functions. The data register addresses the top location of the transmit FIFO and the bottom location of the receive FIFO. If it is written, data is written to the transmit FIFO, if it is read, data is read from the receive FIFO. The status registers contain bits to signal interrupt requests, FIFO state, and error conditions.

The three UARTs have the following base addresses:

Port	Base Address
Serial port 1, UART1	0x8001 0000
Serial port 2, UART2	0x8003 0000
Serial port 3, UART3	0x8005 0000

Table 11. Base addresses for SA-1100 UARTs 1, 2, and 3

The registers are as follows:

Offset	Name	Description	Access
0x00	UTCR0	UART Control Register 0	R/W
0x04	UTCR1	UART Control Register 1	R/W
0x08	UTCR2	UART Control Register 2	R/W
0x0C	UTCR3	UART Control Register 3	R/W
0x10	UTCR4	UART Control Register 4 (only UART 2)	R/W
0x14	UTDR	UART Data Register	R/W
0x1C	UTSR0	UART Status Register 0	R/W & RO
0x20	UTSR1	UART Status Register 1	RO

Table 12. UART register block

4.8.4.1 UART Control Register 0 (UTCR0)

Bit	Name	Description
0	PE	Parity Enable 0 - Parity checking on received data and parity generation on transmitted data is enabled. 1 - Parity checking on received data and parity generation on transmitted data is disabled.
1	OES	Odd/Even Parity Select 0 - Odd parity checking/generation selected. 1 - Even parity checking/generation selected.
2	SBS	Stop Bit Select 0 - one stop bit per frame. 1 - two stop bits per frame. Note: the receiver is not affected by SBS, it always checks for one stop bit.
3	DSS	Data Size Select 0 - 7 bit data 1 - 8 bit data (in 7-bit mode data is right justified in the FIFO entries, MSBs are zero filled/ignored)
4	SCE	Sample Clock Enable 0 - on-chip baud rate generator 1 - external sample clock input via GPIO pin, for details see [6]
5	RCE	Receive Clock Edge Select use rising or falling edge of external sample clock, for details see [6]
6	TCE	Transmit Clock Edge Select use rising or falling edge of external sample clock, for details see [6]
7	-	Reserved

Table 13. UART Control Register 0

4.8.4.2 UART Control Register 1 and 2 (UTCR1 and UTCR2)

Control registers 1 and 2 are used to program the baud rate. Control register 1 (UTCR1) contains the upper four bits BRD[11:8] and UTCR2 contains the lower eight bits BRD[7:0] of the 12-bit baud rate divisor field BRD[11:0] which is used to select the baud rate of the UART. The possible 4096 values allow for baud rates ranging from 56.24 bps up to 230.4 Kbps.

The relationship between the resulting baud rate and the BRD value to be programmed into UTCR1 and UTCR2 is given by the following two formulas:

$$BaudRate = \frac{3.6864\times10^{6}}{16 \bullet (BRD + 1)}$$

$$BRD = \frac{3.6864\times10^{6}}{16 \bullet BaudRate} - 1$$

These are the values for setting two frequently used baud rates:

9600 baud: $BRD = \frac{3.6864\times10^{6}}{16 \bullet 9600} - 1 = 23_{d} = 0x17_{h}$, thus UTCR2 = 0x17, UTCR1 = 0x0

115200 baud: $BRD = \frac{3.6864\times10^{6}}{16 \bullet 115200} - 1 = 1_{d} = 1_{h}$, thus UTCR2 = 0x01, UTCR1 = 0x0

Bits 4-7 of UTCR1 are reserved, writes are ignored, reads return zeros.

4.8.4.3 UART Control Register 3 (UTCR3)

UTCR3 is an eight-bit register that mainly contains the bits to enable the receiver and the transmitter and the related interrupts.

Bit	Name	Description
0	RXE	Receiver Enable Enable/disable UART receive operation. When the receiver is disabled (RXE=0), control of its RXD pin is transferred to the PPC. **Before setting RXE, all other control bits, including the transmit bits, have to be set**. If RXE is cleared to zero while the UART is receiving data, the receive operation is stopped, the bits within the receive serial shifter and all entries within the receive FIFO are reset, while all other control/status/flag bits remain intact.
1	TXE	Transmitter Enable Enable/disable UART transmit operation. When the transmitter is disabled (TXE=0), control of its TXD pin is transferred to the PPC. **Before setting TXE, all other control bits, including the receive bits, have to be set**. If TXE is cleared to zero while the UART is transmitting data, the transmit operation is stopped, remaining bits within the transmit serial shifter and all entries within the transmit FIFO are reset, while all other control/status/flag bits remain intact.
2	BRK	Break If the break control bit is set, a break is transmitted by forcing the transmit pin low. For more information see [6]
3	RIE	Receive Interrupt Enable This bit is used to enable/disable both the receive FIFO service request interrupt and the receiver idle interrupt. If RIE=0 these interrupts are masked and the state of the receive FIFO service request (RFS) and receiver idle (RID) status bits is ignored by the interrupt controller. This bit only controls the generation of the interrupt request, it does not affect the state of RFS and RID and generation of the receive FIFO DMA request which is asserted any time RFS=1.
4	TIE	Transmit Interrupt Enable The TIE bit is used to mask or enable the transmit FIFO service request interrupt. If TIE=0 the interrupt is masked and the state of the transmit FIFO service request (TFS) bit is ignored by the interrupt controller. Again the bit does not affect the state of TFS or the generation of the transmit FIFO DMA request.
5	LBM	Loop Back Mode When LBM=0 the UART operates normally, with independent transmit and receive data paths. When LBM=1 the output of the transmit serial shifter is directly connected to the input of the receive serial shifter and control of the RXD and TXD pins is transferred to the PPC.
6-8	-	Reserved

Table 14. UART Control Register 3

Only TXE and RXE are reset to a known state (0) to ensure that the UARTs are disabled after a reset, while the state of all other UART control bits is unknown after reset. Note that before enabling transmitter or receiver the sticky status bits (see 4.8.4.5 “UART Status Register 0 (UTSR0)“) have to be cleared !

Note: if one of the two engines (transmit or receive) is disabled, then the corresponding interrupts should be disabled. When working with the IR port I had some cases where I got receiver interrupts even though the receiver was disabled !

4.8.4.4 UART Data Register (UTDR)

The UART data register is an eight-bit register corresponding to both the top entry of the transmit FIFO and the bottom entry of the receive FIFO.

If the register is read, the lower 8 bits of the bottom entry of the 11-bit wide receive FIFO are returned. As data enters the top of the receive FIFO, bits 8-10 are used to store error conditions which occurred during reception of the

corresponding data entry. These error bits are transferred down the FIFO together with the data entry that caused the error. When a data entry reaches the bottom of the receive FIFO, the three error flags are transferred to the parity error (PRE) flag, the framing error (FRE) flag, and the receiver overrun (ROR) flag within UART status register 1. These flags can be read before reading the receiver FIFO to determine any errors in the next byte to be read. If any of the error flags is set within the bottom four entries of the receive FIFO, the error in FIFO (EIF) flag bit is set. This causes an interrupt to be generated and receive FIFO DMA requests to be disabled to allow manually emptying the FIFO by checking the error flags in the status register before reading the FIFO entries. After removing an entry the EIF flag should be checked to see if any errors remain, and the procedure repeated until all erroneous entries have been flushed from the FIFO. Then EIF is automatically cleared and DMA requests are re-enabled.

When UTDR is written the top-most entry of the 8-bit transmit FIFO is accessed, causing the data to be transferred down to the lowest location within the transmit FIFO which does not yet contain valid data.

4.8.4.5 UART Status Register 0 (UTSR0)

Status Register 0 contains bits which signal interrupt requests to the interrupt controller. All can be read, some of them are read/write bits and have to be cleared by software by writing ones to them (called sticky status bits, while read-only bits, which are set and cleared by hardware, are referred to as flags). As usual, writing zeros to status bits has no effect, flags are not affected by any write operation.

Bit	Name	Access	Description
0	TFS	RO	Transmit FIFO Service Request Flag TFS is set any time the transmit FIFO has four or fewer entries of valid data and is cleared when it has five or more entries of valid data. When the bit is set, a DMA service request is made, as well as an interrupt request, if TIE=1. Then, after DMA or CPU (via programmed I/O) have filled the FIFO to contain at least five entries, TFS, DMA request, and interrupt request are automatically cleared.
1	RFS	RO	Receive FIFO Service Request Flag RFS is set when the receive FIFO contains between five and eight entries of valid data. When the bit is set, a DMA service request is made, as well as an interrupt request, if RIE=1. As described previously, DMA has to be configured to a burst size of four words, alternatively four entries may be read using programmed I/O without checking. Then the receive FIFO not empty (RNE) flag must be polled before each read to see if more data remains. If the FIFO has been emptied so that at least five locations are available the RFS flag and DMA and interrupt request are cleared.
2	RID	R/W	Receiver Idle Status The RID bit is set when the receiver is enabled (RXE=1), the receive FIFO contains at least one entry of data and the receiver has been idle for three frame periods. If RIE=1 an interrupt request is made when RID is set.
3	RBB	R/W	Receiver Begin of Break Status This bit is set if the begin of a break is detected, i.e. when the receive line is held low for one frame duration. When RBB is set, an interrupt is signalled and a single null byte is placed in the FIFO. Additionally the framing error bit is set and all subsequent null frames with framing errors are ignored. RBB has to be cleared by software and can not be set again until the receiver end of break status (REB) bit has been set. This interlock is cleared when REB is set, when RXE is cleared or the SA-1100 is reset.
4	REB	R/W	Receiver End of Break Status The REB bit is set when the RBB interlock is set and the receiver detects a rising edge on the receive pin. When REB is set an interrupt is signalled and the RBB interlock is cleared so that any future data frames will be stored in the receive FIFO. After the bit is cleared it will not be set again until the RBB bit is set once again.

Table 15. UART Status Register 0

Bit	Name	Access	Description
5	EIF	RO	Error in FIFO The EIF flag is set if any of the error bits is set within the bottom four entries of the receive FIFO and is cleared if there are no remaining errors within the bottom four entries. If EIF is set, an interrupt request is made and receive DMA requests are disabled until EIF is cleared again. The source of the error can be found by reading the error flags in status register 1. When EIF has been cleared after removing the bottom four FIFO entries via programmed I/O the interrupt request is cleared and DMA is re-enabled.
6-8	-	-	Reserved

Table 15. UART Status Register 0

The reset state of all writable status bits is unknown and must be explicitly cleared by writing a one to them before enabling the UART.

4.8.4.6 UART Status Register 1 (UTSR1)

Status Register 1 contains read-only flags that indicate the current status of the UART:

Bit	Name	Description
0	TBY	Transmitter Busy Flag TBY is set when the transmitter is processing data for transmission (the serial shifter contains data) and is cleared when the transmitter is idle or is disabled (TXE=0).
1	RNE	Receive FIFO Not Empty RNE is set whenever the receive FIFO contains one more bytes of valid data and is cleared when it does not contain any more data. This bit can be polled when reading the receive FIFO using programmed I/O after DMA or interrupt requests.
2	TNF	Transmit FIFO Not Full Flag TNF is set whenever the transmit FIFO contains one or more empty entries and is cleared when the transmit FIFO is completely full. This bit can be polled when using programmed I/O to fill the transmit FIFO over its half-way mark.
3	PRE	Parity Error Flag The PRE flag is set if the bottom entry in the receive FIFO contains a parity error.
4	FRE	Framing Error Flag FRE is set when the stop bit of the bottom entry in the receive FIFO was zero instead of one when the frame was received.
5	ROR	Receiver Overrun Flag ROR is set within the top entry of the receive FIFO whenever an overrun occurs, i.e. data is received while the FIFO is full. This tag travels along with the last byte received before the overrun occurred, as it moves through the FIFO. When this data entry reaches the bottom FIFO entry the error tag is transferred to the ROR bit.
6-7	-	Reserved

Table 16. UART Status Register 1

Note: I found that, at least in some cases, TBY is not set immediately after transmission of a character has been started. I noticed this while working with test code for the IrDA implementation: as IrDA is half-duplex I started the transmission and then went into a busy loop checking the Transmitter Busy Flag to find the end of the transmission before switching to receive mode. The test for the TBY was the first code executed after starting the transmission and apparently this does not leave enough time for the TBY to be set. Hence, if the TBY flag has to be checked immediately after starting the transmission, some sort of a delay has to be inserted to make sure that the flag is actually set before it is tested the first time. I made some tests using an empty for-loop to generate the delay. It turned out that the needed time varied from a loop count of about 5 to 100. Generally a loop count of 150-300 should be sufficient.

See chapter 10. for a description on how to detect when the transmission is finished.

4.8.5 UART Declarations and Functions

Macros and declarations to use the UART channels:

```
/*
 *      Description:
 *          Type declarations for use with serial ports (UARTs) on the Badge
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#ifndef util_serial_h
#define util_serial_h

#include <util/util_ringbuf.h>
#include <stdio.h>


/* serial port base addresses */
#define UART1_BASE  0x80010000     /* UART 1 base address */
#define UART2_BASE  0x80030000     /* UART 2 (Infrared port low speed) base */
#define UART3_BASE  0x80050000     /* UART 3 base (used by Angel)*/

#define SDLC_BASE   0x80020060     /* SDLC base address */


/* SDLC regs/values offsets from base address */
#define SDCR0   0x00    /* SDLC Control Register 0 */
#define SDCR1   0x04    /* SDLC Control Register 1 */
#define SDCR2   0x08    /* SDLC Control Register 2 */
#define SDCR3   0x0C    /* SDLC Control Register 3 */
#define SDCR4   0x10    /* SDLC Control Register 4 */
#define SDDR    0x18    /* SDLC Data Register */
#define SDSR0   0x20    /* SDLC Status Regiser 0 */
#define SDSR1   0x24    /* SDLC Status Regiser 1 */

/* SDLC control register bits */
#define SDCR0_SUS   0x01    /* SDLC/UART Select */


/* UART register bits */

/* UART control register 0 bits */
#define UTCR0_PE    0x01    /* Parity Enable */
#define UTCR0_OES   0x02    /* Odd/Even Parity Select */
#define UTCR0_SBS   0x04    /* Stop Bit Select */
#define UTCR0_DSS   0x08    /* Data Size Select */
#define UTCR0_SCE   0x10    /* Sample Clock Enable */
#define UTCR0_RCE   0x20    /* Receive Clock Edge Select */
#define UTCR0_TCE   0x40    /* Transmit Clock Edge Select */

/* UART control register 3 bits */
#define UTCR3_RXE   0x01    /* Receiver Enable */
#define UTCR3_TXE   0x02    /* Transmitter Enable */
#define UTCR3_BRK   0x04    /* Break */
#define UTCR3_RIE   0x08    /* Receive FIFO Interrupt Enable */
#define UTCR3_TIE   0x10    /* Transmit FIFO Interrupt Enable */
#define UTCR3_LBM   0x20    /* Loop Back Mode */

/* UART interrupt status bits (status register 0) */
#define UTSR0_TFS   0x01    /*  transmit fifo service request  */
```

Listing 12. UART macros and declarations [file util\util_serial.h]

```
#define UTSR0_RFS   0x02    /*  receive fifo service request  */
#define UTSR0_RID   0x04    /*  receiver idle  */
#define UTSR0_RBB   0x08    /*  receiver begin of break  */
#define UTSR0_REB   0x10    /*  receiver end of break  */
#define UTSR0_EIF   0x20    /*  error in fifo  */

/* UART line status bits. (status register 1) */
#define UTSR1_TBY   0x01    /*  transmitter busy flag  */
#define UTSR1_RNE   0x02    /*  receiver not empty (LSR_DR)  */
#define UTSR1_TNF   0x04    /*  transmit fifo non full  */
#define UTSR1_PRE   0x08    /*  parity read error (LSR_PE)  */
#define UTSR1_FRE   0x10    /*  framing error (LSR_FE)  */
#define UTSR1_ROR   0x20    /*  receive fifo overrun (LSR_OE)  */



/* UART register offsets from base address */
#define UTCR0       0x00    /* UART Control Register 0 */
#define UTCR1       0x04    /* UART Control Register 1 */
#define UTCR2       0x08    /* UART Control Register 2 */
#define UTCR3       0x0C    /* UART Control Register 3 */
#define UTCR4       0x10    /* UART Control Register 4 */
#define UTDR        0x14    /* UART Data Register */
#define UTSR0       0x1C    /* UART Status Register 0 */
#define UTSR1       0x20    /* UART Status Register 1 */


/* macros for UART register access */

// enable receiver, transmitter or both
#define SER_UART_ENABLE_R(base)     (REG(base, UTCR3) |= UTCR3_RXE)
#define SER_UART_ENABLE_T(base)     (REG(base, UTCR3) |= UTCR3_TXE)
#define SER_UART_ENABLE_TR(base)    (REG(base, UTCR3) |= (UTCR3_RXE | UTCR3_TXE))

// disable receiver, transmitter or both
#define SER_UART_DISABLE_R(base)    (REG(base, UTCR3) &= ~UTCR3_RXE)
#define SER_UART_DISABLE_T(base)    (REG(base, UTCR3) &= ~UTCR3_TXE)
#define SER_UART_DISABLE_TR(base)   (REG(base, UTCR3) &= ~(UTCR3_RXE | UTCR3_TXE))

// enable receiver/transmitter interrupt
#define SER_UART_ENABLE_RI(base)    (REG(base, UTCR3) |= UTCR3_RIE)
#define SER_UART_ENABLE_TI(base)    (REG(base, UTCR3) |= UTCR3_TIE)
#define SER_UART_ENABLE_TRI(base)   (REG(base, UTCR3) |= (UTCR3_TIE | UTCR3_RIE))

// disable receiver/transmitter interrupt
#define SER_UART_DISABLE_RI(base)   (REG(base, UTCR3) &= ~UTCR3_RIE)
#define SER_UART_DISABLE_TI(base)   (REG(base, UTCR3) &= ~UTCR3_TIE)
#define SER_UART_DISABLE_TRI(base)  (REG(base, UTCR3) &= ~(UTCR3_TIE | UTCR3_RIE))

// enable/disable loopback mode
#define SER_UART_ENABLE_LOOPBACK(base)  (REG(base, UTCR3) |= UTCR3_LBM)
#define SER_UART_DISABLE_LOOPBACK(base) (REG(base, UTCR3) &= ~UTCR3_LBM)

// select UART or SDLC operation for serial port 1
#define SER_UART_UART1_SEL_UART (REG(SDLC_BASE,SDCR0) |= SDCR0_SUS)
#define SER_UART_UART1_SEL_SDLC (REG(SDLC_BASE,SDCR0) &= ~SDCR0_SUS)

// query receiver/transmitter status
#define SER_UART_REC_NOT_EMPTY_Q(base) (REG(base,UTSR1) & UTSR1_RNE)
#define SER_UART_TRANSM_NOT_FULL_Q(base) (REG(base,UTSR1) & UTSR1_TNF)
#define SER_UART_TRANSM_BUSY_Q(base) (REG(base,UTSR1) & UTSR1_TBY)

// completely shut down the UART / clear sticky status bits
#define SER_UART_SHUTDOWN(BASE) (REG(base, UTCR3) = 0x0)
#define SER_UART_CLEAR_STATUS_BITS_ALL(base) (REG(base, UTSR0) = 0xFF)
```

Listing 12. UART macros and declarations [file util\util_serial.h]

```
#define SER_UART_CLEAR_STATUS_BITS_RBB(base) (REG(base, UTSR0) = UTSR0_RBB)
#define SER_UART_CLEAR_STATUS_BITS_REB(base) (REG(base, UTSR0) = UTSR0_REB)
#define SER_UART_CLEAR_STATUS_BITS_RID(base) (REG(base, UTSR0) = UTSR0_RID)

// write a byte to the UART data register
#define SER_UART_PUT_BYTE_DIRECT(base, ch) = (REG(base, UTDR) = (ch))

// defines for the UART init function ser_UartInit
#define SER_INIT_PARITY_DIS  0
#define SER_INIT_PARITY_EN   1

#define SER_INIT_PARITY_ODD  0
#define SER_INIT_PARITY_EVEN 1

#define SER_INIT_ONE_STOP_BIT 0
#define SER_INIT_TWO_STOP_BIT 1

#define SER_INIT_DATA_SIZE_7  0
#define SER_INIT_DATA_SIZE_8  1

// currently only these two are defined, format UTCR1<<8|UTCR2
#define SER_INIT_BAUD_9600   0x0017
#define SER_INIT_BAUD_115200 0x0001

#define SER_INIT_RECEIVE_INT_DIS  0
#define SER_INIT_RECEIVE_INT_EN   1

#define SER_INIT_TRANSMIT_INT_DIS 0
#define SER_INIT_TRANSMIT_INT_EN  1


/*
 * Function: ser_UartPutByteHex
 *  Purpose: transmit a byte in hex representation
 *
 * Parameters:
 *      Input: base             base address of UART to be used
 *             ch               the byte to be written
 *
 *    Returns: void
 *
 * The function writes the hexadecimal representation (‘dd ‘ where ‘d’ are
 * hex digits) of the given byte to the UART.
 * (intended and used mainly for debug purposes, therefore declared as
 * inline for execution speed reasons)
 */
__inline void ser_UartPutByteHex(UI32 base, UC8 ch)
{
    char buf[4];
    sprintf(buf, “%2.2x “, ch);

    while(!SER_UART_TRANSM_NOT_FULL_Q(base))
        ;
    REG(base, UTDR) = buf[0];
    while(!SER_UART_TRANSM_NOT_FULL_Q(base))
        ;
    REG(base, UTDR) = buf[1];
    while(!SER_UART_TRANSM_NOT_FULL_Q(base))
        ;
    REG(base, UTDR) = buf[2];
}


#endif
```

Listing 12. UART macros and declarations [file util\util_serial.h]

Functions to use the UARTS:

```
/*
 *      Description:
 *          Functions to access the serial ports (UARTs) on the Badge.
 *          SDLC (port 1) and MCP/SSP (serial port 4) are not yet implemented)
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#include    <stdlib.h>
#include    <string.h>

#include <util/util_serial.h>


/*
 * Function: ser_UartInit
 *  Purpose: Initialize a UART port
 *
 * Parameters:
 *      Input: base             base address of UART to be initialized
 *             baud             baud rate to be used
 *             parity_en        en/disable parity check and generation
 *             parity_sel       select odd or even parity
 *             stop             one or two stop bits
 *             data_size_sel    7 or 8 data bits
 *             receive_int      en/disable receive interrupt
 *             transmit_int     en/disable transmit interrupt
 *
 *    Returns: void
 *
 * The function initializes all control registers of the given UART, clears
 * the sticky status bits, but doesn not yet enable the receiver and
 * transmitter.
 * The value to be supplied for the parameter baud is BRD[11..0], see
 * constants in the include file (also for the other parameters).
 */
void ser_UartInit(UI32 base, UI16 baud, int parity_en,
               int parity_sel, int stop, int data_size_sel,
               int receive_int, int transmit_int)
{

    // serial port 1 contains UART and SDLC, thus UART has to be selected
    if(base == UART1_BASE)
    {
        SER_UART_UART1_SEL_UART;
    }

    // disable the transmitter and receiver and clear any status bits
    REG(base, UTCR3) = 0x00000000;
    SER_UART_CLEAR_STATUS_BITS_ALL(base);

    // control register 0
    // PE  = 0/1 ; parity disabled/enabled
    // OSE = 0/1   ; dont care about even/odd parity
    // SBS = 0/1 ; one stop bit
    // DSS = 0/1 ; 7/8 data bits
    // SCE = 0   ; on chip baud rate generator
    // RCE = X   ; reveive edge unused
    // TCE = X   ; transmit edge unused
    REG(base, UTCR0) = (data_size_sel<<3)|(stop<<2)|(parity_sel<<1)|parity_en;
```

Listing 13. Functions to use the UARTs [file util_serial.c]

```
    // set the BRD fields in control registers 1 and 2
    REG(base, UTCR1) = baud >> 8;
    REG(base, UTCR2) = (baud & 0xff);


    // control register 3
    // RXE = 0 ; receive enable
    // TXE = 0 ; transmit enable
    // BRK = 0  ; break disabled
    // RIE = 0/1    ; receive interrupt
    // TIE = 0/1    ; transmit interrupt
    REG(base, UTCR3) =  0x00000000 | (receive_int<<3) | (transmit_int<<4);

}

/* Polled I/0 */

/*
 * Function: ser_UartGetByte
 *  Purpose: read a byte from the given UART channel
 *
 * Parameters:
 *      Input: base             base address of UART to be used
 *
 *    Returns: the read byte
 *
 * The function reads a byte from the UART data register until
 * having waited in a busy loop until there is a valid entry in the
 * receive FIFO (i.e. the flag RNE becomes 1)
 */
unsigned char ser_UartGetByte(UI32 base)
{

    // wait in a busy loop until there is a byte in the receive FIFO
    while(!SER_UART_REC_NOT_EMPTY_Q(base))
        ;

    // Read the byte
    return REG(base, UTDR);
}


/*
 * Function: ser_UartPutByte
 *  Purpose: transmit a byte using a given UART channel
 *
 * Parameters:
 *      Input: base             base address of UART to be used
 *             ch               the byte to be written
 *
 *    Returns: void
 *
 * The function writes the given byte to the UART data register until
 * having waited in a busy loop until there is a free entry in the
 * transmit FIFO (i.e. the flag TNF becomes 1)
 */
void ser_UartPutByte(UI32 base, unsigned char ch)
{

    // wait in a busy loop until transmit FIFO not full
    while(!SER_UART_TRANSM_NOT_FULL_Q(base))
        ;
```

Listing 13. Functions to use the UARTs [file util_serial.c]

```
    // put the character into the transmit FIFO
    REG(base, UTDR) = ch;
}


/*
 * Function: ser_UartPutStringDirect
 *  Purpose: transmit a null-terminated string on the given UART by
 *           directly writing to the transmit FIFO
 *
 * Parameters:
 *      Input: base             base address of UART to be used
 *             string           the string to be output
 *
 *    Returns: void
 *
 * The function writes the string to the transmit FIFO of the given
 * UART. For each character to transmit it waits until there is
 * space in the FIFO.
 */
void ser_UartPutStringDirect(UI32 base, unsigned char* string)
{
    int i;
    int len = strlen((char*)string);
    for(i=0;i<len;i++)
    {
        while(!SER_UART_TRANSM_NOT_FULL_Q(base))
            ;

        // put the character into the transmit FIFO
        REG(base, UTDR) = string[i];
    }
}


/* Interrupt driven I/O */

/*
 * Function: ser_UartPutStringInt
 *  Purpose: transmit a null-terminated string on the given UART using
 *           a RingBuffer
 *
 * Parameters:
 *      Input: base             base address of UART to be used
 *             tx_buf           the RingBuffer to be used for the output
 *             string           the string to be output
 *
 *    Returns: 0                o.k.
 *             -1               not enough space in the RingBuffer
 *
 * The function writes the string to the given ringbuffer (provided
 * it has enough space) and then activates the transmit interrupt on
 * the given UART to start the transmission
 */
int ser_UartPutStringInt(UI32 base, RingBuffer* tx_buf, unsigned char* string)
{
    int i;
    int len = strlen((char*)string);
    if(len<=ringbuf_GetSpace(tx_buf))
    {
        for(i=0;i<len;i++)
            ringbuf_WriteByte(tx_buf, string[i]);

        SER_UART_ENABLE_TI(base);
        return 0;
```

Listing 13. Functions to use the UARTs [file util_serial.c]

```
    }
    else
        return 1;
}


/*
 * Function: ser_UartPutStringBlocking
 *  Purpose: transmit a null-terminated string on the given UART using
 *           a RingBuffer and wait until there is enough space in the
 *           buffer.
 *
 * Parameters:
 *      Input: base             base address of UART to be used
 *             tx_buf           the RingBuffer to be used for the output
 *             string           the string to be output
 *
 *    Returns: void
 *
 * The function waits until there is enough space in the RingBuffer, then
 * writes the string to the buffer and activates the transmit interrupt on
 * the given UART to start the transmission
 */
void ser_UartPutStringBlocking(UI32 base, RingBuffer* tx_buf,
                               unsigned char* string)
{
    int i;
    int len = strlen((char*)string);
    while(len>ringbuf_GetSpace(tx_buf))
        ;
    for(i=0;i<len;i++)
        ringbuf_WriteByte(tx_buf, string[i]);

    SER_UART_ENABLE_TI(base);
}


/*
 * Function: ser_UartPutNewLineInt
 *  Purpose: transmit a CR/LF on the given UART using a RingBuffer
 *
 * Parameters:
 *      Input: base             base address of UART to be used
 *             tx_buf           the RingBuffer to be used for the output
 *
 *    Returns: 0                o.k.
 *             -1               not enough space in the RingBuffer
 *
 * The function writes the CR/LF bytes to the given ringbuffer (provided
 * it has at least two free positions) and then activates the transmit
 * interrupt on the given UART to start the transmission
 */
int ser_UartPutNewLineInt(UI32 base, RingBuffer* tx_buf)
{
    if(ringbuf_GetSpace(tx_buf)>=2)
    {
        ringbuf_WriteByte(tx_buf, 0x0a);
        ringbuf_WriteByte(tx_buf, 0x0d);
        SER_UART_ENABLE_TI(base);
        return 0;
    }
    else
        return -1;
}
```

Listing 13. Functions to use the UARTs [file util_serial.c]

5. Infrared Communication

Serial port 2 of the SA-1100 is intended for infrared communication. It provides two different units, one for the original IrDA standard for speeds up to 115200 baud (referred to as SIR mode in the following), the other one for the newer 4.0Mbps standard (referred to as FIR mode). Both units have a distinct set of control registers, but they share the same pins for transmission/receiving and the same interrupt and cannot be operated at the same time. Each unit consists of a bit encoder/decoder and a serial to parallel data engine. SIR mode is based on a UART (UART 2) while the engine for the high speed mode is a high speed serial to parallel (HSSP) receiver-transmitter. To support a variety of IrDA transceivers, both the transmit and the receive data pins can be configured to use either normal or inverted data. The pins are designed to be directly connected to infrared transceiver modules. After the design for SmartBadge 3 had already been finished it turned out that only a very small percentage of the intended IR transceiver modules actually worked at the stated and required voltage specifications. Therefore a small daughtercard was added to convert between the original pin layout on the SmartBadge board and the (different) layout of the new transceiver module. Another severe problem was that the original SA-1100 has a bug in the SIR modulation unit which makes it unusable for transmission (although the receive operation works properly). In this chapter I will first describe some small modifications to the daughtercard to simplify watching the communication by using a logic analyzer and to allow SIR mode which was not supported in the default configuration of the daughtercard. Then I describe the SIR unit, including a method to perform SIR software modulation in order to circumvent the previously mentioned SA-1100 bug. After that I describe the FIR unit, followed by a description of the setup and environment I used to debug the infrared communication. The infrared support code can be found in Appendix A "Infrared support code", some example programs making use of this support code are listed in Appendix B "Example Programs". The user of the IrDA interface for higher level protocols will be described in subsequent chapters.

5.1 Hardware Modifications to the SmartBadge

5.1.1 Modifications to Observe Communication via a Logic Analyzer

As infrared communication was not working properly at the beginning of my experiments - although I thought that I had configured everything properly - I looked for a way to observe the generated signal. By connecting two wires to the IR daughtercard (on paths leading to pins 8 and 9, see Figure 16 and Figure 17) I was able to attach a logic analyzer to the transmit and the receive pins respectively. (Logic analyzer) Ground was connected to the daughtercard shield. Using the logic analyzer I found that in SIR mode the correct bit pattern is generated by the SA-1100, but at a frequency that was too high. It seems that the SIR modulation unit is incorrectly clocked by the FIR clock. This explained why I could not receive anything from the Badge on other SIR devices I had used (this included various notebooks with IrDA ports and a second Badge). Subsequently I learned that this was due to a bug in the implementation of the SA-1100.

5.1.2 Modifications to Allow SIR Mode

The transceiver module that originally had been planned to be used had only one mode pin that allowed a processor to select between SIR and FIR mode, therefore only one GPIO pin (GPIO26) was dedicated to control the transceiver module. The IR transceiver module which is now used has three mode pins, one to select between SIR and FIR and two to select between various power modes (Shutdown, 1/3, 2/3, or full power).

The truth table for this new module is shown in Table 17. To fit the new module into the existing Badge design (with a different pin-layout for the module and only one GPIO pin available to be used for mode select) a small daughter-

card, which is plugged into a connector at the position of the original transceiver module, was added. On that card the mode pin MD1 is hardwired to ground. This disables all combinations which are greyed in the truth table. Mode pin MD0 now switches between “Shutdown” and “Full Distance Power”, while FIR_SEL selects between SIR and FIR mode.

MD0	MD1	FIR_SEL	RX Function	TX Function
1	0	X	Shutdown	Shutdown
0	0	0	SIR	Full Distance Power
0	1	0	SIR	2/3 Distance Power
1	1	0	SIR	1/3 Distance Power
0	0	1	MIR/FIR	Full Distance Power
0	1	1	MIR/FIR	2/3 Distance Power
1	1	1	MIR/FIR	1/3 Distance Power

Table 17. HSDL-3600 Transceiver Control Truth Table

The design of the IR daughtercard allows two configurations:

- Transmission mode fixed to FIR, select between Full Distance Power and Shutdown via GPIO 26 (default configuration for the SmartBadge)
- Fixed to Full Distance Power, select between SIR and FIR transmission mode via GPIO 26 (requires a small hardware modification).

Figure 16 shows the circuit diagram of the IR daughtercard in the default configuration. FIR_SEL is connected to VCC via a 10k pull-up resistor (R2) and MD0 is connected to the GPIO pin via the 0-ohm resistor R1 (which acts just as a jumper), resistor R3 is not mounted. As mentioned, this fixes the mode to be FIR and allows the processor to select between Full Distance Power and Shutdown - SIR mode is not available, but due to the bug in the early versions of the SA-1100 this mode was not usable in any case.

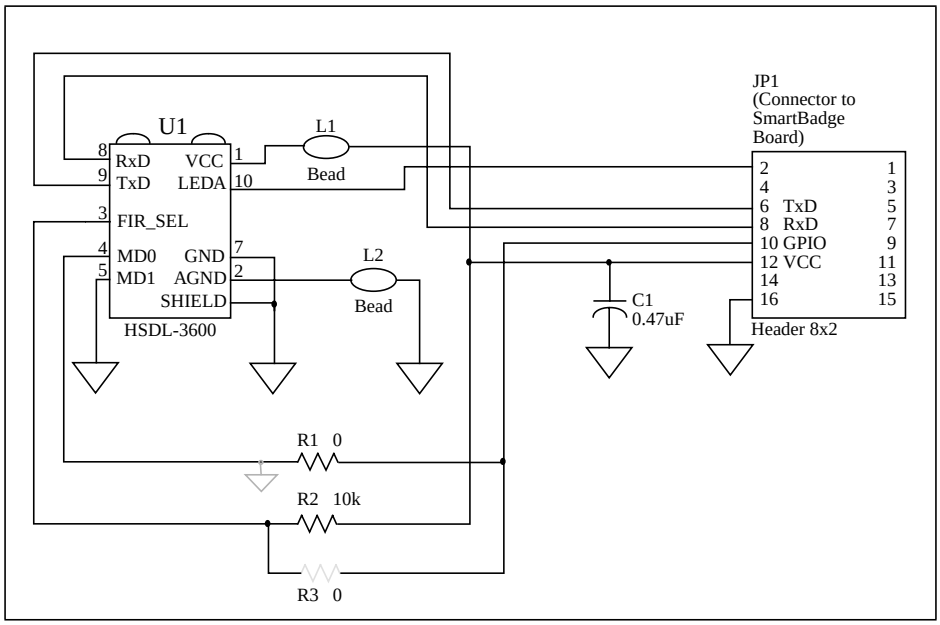


Figure 16. IR daughtercard circuit diagram (default configuration)

One of my tasks was to implement IrDA communication to provide IP network connectivity to a badge via an HP NetbeamIR access point[1]. Although from the point of view of speed and power consumption FIR mode is clearly

preferable, unfortunately the IrDA protocol suite (in particular IrLAP) requires that the initial communication take place in SIR mode at 9600 baud (with the possibility to negotiate a higher speed afterwards). So I tried to get around the SA-1100 bug with software modulation as will be described later, but this still required a modification to the transceiver module to operate in SIR mode. The necessary changes are as follows:

As can be seen from the truth table, the only way to allow both SIR **AND** FIR mode is by setting mode pin MD0 to zero and use pin FIR_SEL to switch between the two transmission modes. This can be done by removing the 0-ohm resistor R1 and adding a 0-ohm resistor R3, and connecting MD0 to ground. This modification is shown greyed in Figure 16. Now MD0 is fixed to zero and the level of FIR_SEL can be set via GPIO 26 (the remaining pull-up resistor R2 doesn't change the function, but could be removed to save power -- when GPIO 26 is driven low for SIR mode).

Figure 17 shows the topside layout for the original and the modified IR daughtercard. The modification is done by moving the 0-ohm resistor from position R1 to R3 (step 1 in the figure) and grounding pin MD0 (i.e., pin 4 of the IR transceiver U1, e.g. by connecting the now free pad of R1 to the nearby groundplane, which is connected to pins 5 and 7 of U1 - step 2 in the figure). After this modification the transceiver module is tied to full power mode, but can now be switched between SIR and FIR mode via GPIO 26. It should also be noted that the transceiver can no longer be powered down, this causes a small waste of power as the detector is always enabled.

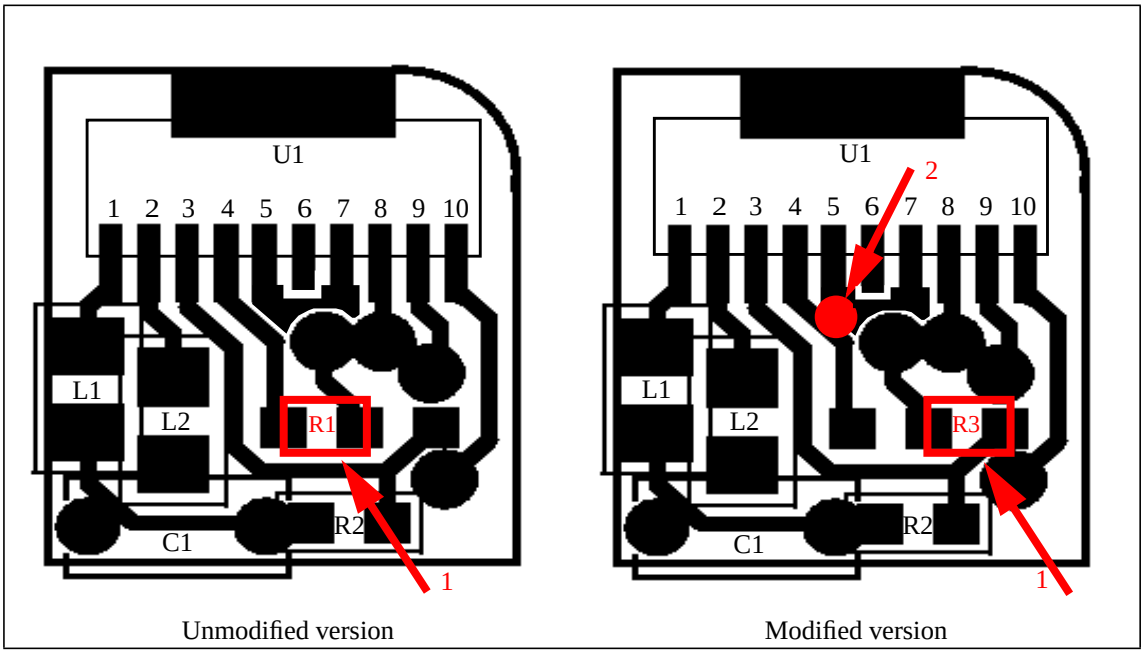


Figure 17. IR Daughtercard Modification

5.2 SIR Mode

Low speed IrDA transmission uses the Hewlett-Packard Serial Infrared standard (SIR) for bit encoding and UART 2 as the serial engine. Following reset both the UART and the HSSP engine are disabled and control of the serial port 2 pins is given to the PPC which configures them as inputs. If IrDA transmission is not needed, UART 2 can be enabled while disabling the HP-SIR encoder, thus serial port 2 can be used as another general purpose serial port.

SIR modulation is used for baud rates up to 115.2 Kbps. Logic zero is represented by a pulse of light which is either 3/16 of the bit time or 1.6 µs wide (1.6µs being 3/16 of the bit time of the maximum bit rate of 115.2 Kbps).

1. This access point utilizes IrLAN (described in chapter 8.) to allow IR frames to and from a device to be bridged to a wired ethernet.

The rising edge of the pulse constitutes the start of the bit time. Logic one is represented by the absence of light pulses. As the start bit is zero, each serial frame begins with a pulse allowing the receiver to detect the beginning of the frame and synchronize the receiver clock. Note that as with a normal serial frame the data byte is transmitted starting with the least significant bit (LSB). Figure 18 shows an example of HP-SIR modulation of the ASCII character ‘F’, represented by the byte 01000110b.

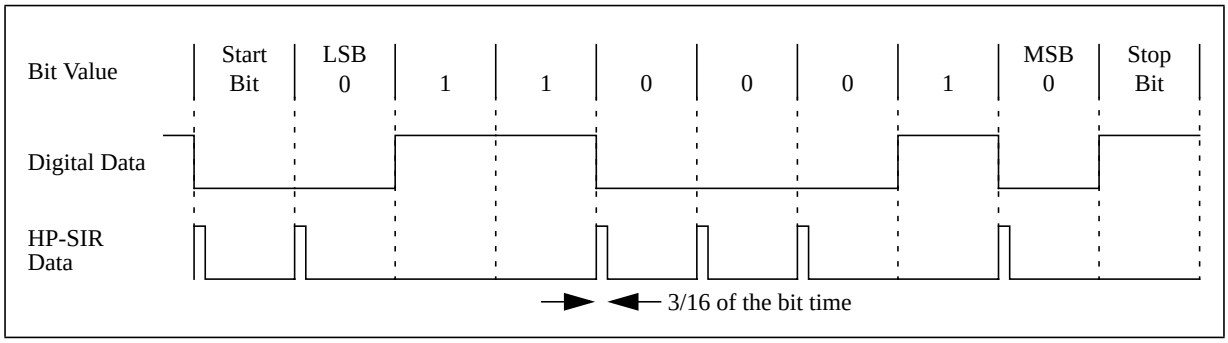


Figure 18. HP-SIR Modulation Example

The required frame format is eight data bits, one stop bit, and no parity bit.

5.2.1 SIR Using Normal UART Operation on Revised SA-1100

Newer revisions of the SA-1100 don’t have the described bug any longer and thus can be used for SIR communication. Programming UART 2 is exactly the same as for the standard UART 3 (see section 4.8 “UARTs”). However, it contains one additional control register (control register 4) to control HP-SIR modulation. Similarly as in serial port one, UART mode for serial port 2 has to be enabled by clearing bit 0 (ITR - IrDA Transmission Rate) in HSSP control register 0.

5.2.1.1 UART Control Register 4 (UTCR4)

UTCR4 consists of two bits, HP-SIR Enable (HSE) as bit 0 and Low Power Mode (LPM) as bit 1.

HP-SIR Enable (HSE)

When HSE=0, HP-SIR modulation is disabled and, if UART 2 operation is enabled in HSSP control register 0, serial port 2 can be used for normal, general purpose serial communication (NRZ encoding). If HSE=1, HP-SIR modulation is enabled, zeros are represented by pulses of 3/16 of the programmed bit width (or 1.6µs wide), while ones are represented by no pulses.

Low Power Mode (LPM)

This bit controls whether zeros are represented by pulses of 3/16 of the chosen bit width or pulses of fixed 1.6µs width. If LPM=0, zeros are encoded as pulses whose width is 3/16 of the bit width programmed via the UART’s baud rate divisor field. When LPM=1 the programmed bit length is ignored for the pulse generation, each pulse has a fixed width of 1.6µs. This minimizes the on-time of the off-chip IR LED transmitter and thus minimizes power consumption.

Note that the UART must be disabled (RXE=TXE=0) when changing the state of either of these two bits.

5.2.1.2 Serial Port 2 Configuration for SIR Mode

To use serial port 2 as a SIR port the following steps have to be done:

- Initialize UART 2 for the desired speed and with parameters 8N1 (eight data bits, no parity, one stop bit).
- Clear bit 0 (ITR) of HSSP Control Register 0 to select SIR modulation/UART engine for port 2 (see section 5.3.7.1 for more information).
- Configure GPIO 26 as output and clear it to zero (selects SIR mode for the transceiver module).

Once all of the above steps are complete, then receive/transmit operation can begin.

5.2.2 SIR via Software Modulation

As mentioned before, the SIR transmission unit of the original SA-1100 cannot be used, but SIR mode is required to initiate communication with the HP NetbeamIR. Therefore I looked for a way to do the required modulation in software. This was possible, because the transmit and receive engines can be enabled and disabled independently. So by enabling the receive engine it was possible to do all receive operation within the UART unit while disabling the transmit engine gave control of the transmit (TXD) pin to the PPC unit which could be used to directly modulate the pin to produce the desired signal.

Some basic facts: as described, zeros are represented by pulses of 3/16 of the bit width while ones are represented by no pulses. At 9600 bps one bit time has a duration of about 104µs. The basic unit therefore is 1/16 of 104µs which is about 6.5µs. If a zero bit is to be transmitted, the pin has to be set for 19.53µs and then be reset for the following 84.6µs. If a one bit is to be transmitted the pin stays reset for the complete 104µs. This concept was implemented using one of the OS Timer channels to generate an interrupt after a timeout depending on the current state.

At the time I was using a version of Angel which had the complete memory configured as non-cacheable. It turned out that in this configuration the achievable interrupt frequency was too slow to support software modulation. After changing the memory configuration to cacheable, SIR transmission using the software modulation was possible. While there were no problems in simple demonstration programs that only transmitted sequences of characters out of a buffer, it turned out though that in my IrDA implementation with lots of other tasks (software timers,...) running at the same time, packets were corrupted in some cases. I was not able to determine the exact reason, but my suspicion is that another interrupt (either a timer interrupt or the Angel communications interrupt used to communicate with the debugger) occurred at an “unlucky” time, thus slightly deferring the OS Timer interrupt responsible for the software modulation. Any delay caused by the occurrence of another interrupt of course completely destroys the bit timing. A fact that supports this suspicion is that the corrupted packets contained sequences of 0xffs - i.e. no pulses for eight bit times - after which the receiver resynchronized and properly received the rest of the packet. I tried to switch off the software timer functionality (except for the core interrupt handler to increment the tick count and to reset the status bit) but that didn’t solve the problem. With additional code it should be possible to delay (i.e. block completely) the timer interrupt while a character is transmitted and only allow interrupts in between but this would make the code more complicated and still not solve the problem of the Angel interrupt. So while software modulation is possible in theory, its practical use seems to be limited to applications where complete control over the interrupts is possible. Another possibility might be to implement the modulation interrupt as an FIQ, but this might require substantial changes to Angel (e.g. regarding the stack setup and allowing FIQs to interrupt an IRQ currently being serviced - as far as I know this is not supported in the standard Angel implementation).

5.3 FIR Mode

For high-speed infrared communication the HSSP (High-Speed Serial to Parallel) unit is used. As with the other serial ports, before enabling the port for high-speed operation, all writable status bits have to first be reset by writing a one to them. Then the intended mode is chosen by setting the control registers. Optionally the HSSP transmit FIFO can be pre-filled with up to 16 values, then the HSSP can be enabled and receive/transmit operation can begin.

In the following sections I will first describe the modulation scheme used for high-speed infrared operation, then the frame format, followed by some more information regarding receive and transmit operation, and the FIFOs. Finally I will explain the HSSP registers.

5.3.1 Infrared High-Speed Modulation

The 4.0Mbps infrared standard uses a different modulation scheme than HP-SIR mode, called 4PPM (four-position pulse) modulation. Two data bits are encoded at a time by placing a single 125ns light pulse within one of four timeslots. The set of four timeslots is called a “chip”. Data is encoded byte by byte, each of which is divided into four nibbles (two-bit pairs) and transmitted LSB first. Figure 19 shows the 4PPM encoding of the four possible 2-bit combinations, Figure 20 shows an example of 4PPM modulation of the byte 10110001b. Note that the byte is taken as it is and divided into the four nibbles. These are encoded and THEN the nibbles are reordered to transmit the least significant (nibble 0) first and the most significant (nibble 3) last.

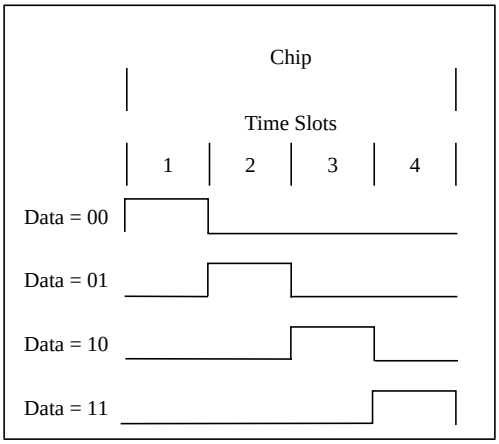


Figure 19. 4PPM Modulation Encoding

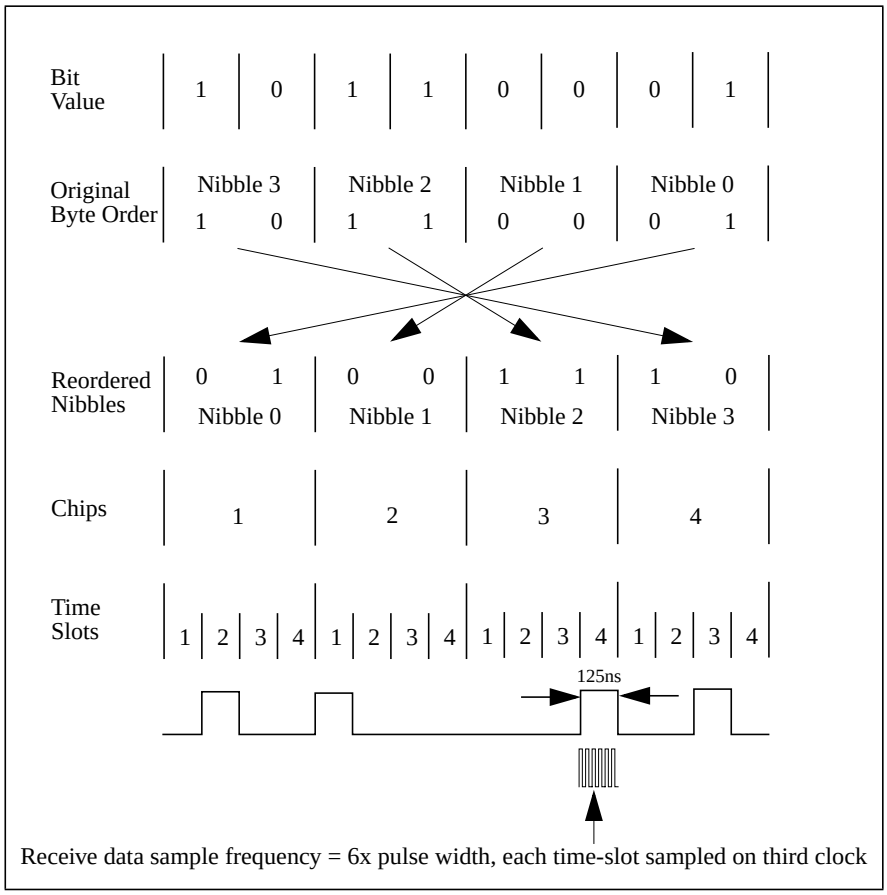


Figure 20. 4PPM Modulation example

5.3.2 HSSP Frame Format

The frame format defined for the infrared high-speed mode is shown in Figure 21. It is derived from the SDLC format (alternatively used in serial port 1), with the following modifications:

- HSSP start/stop flags and CRC are twice as long as in the SDLC standard.
- Instead of only a start flag, a preamble and a start flag are used.

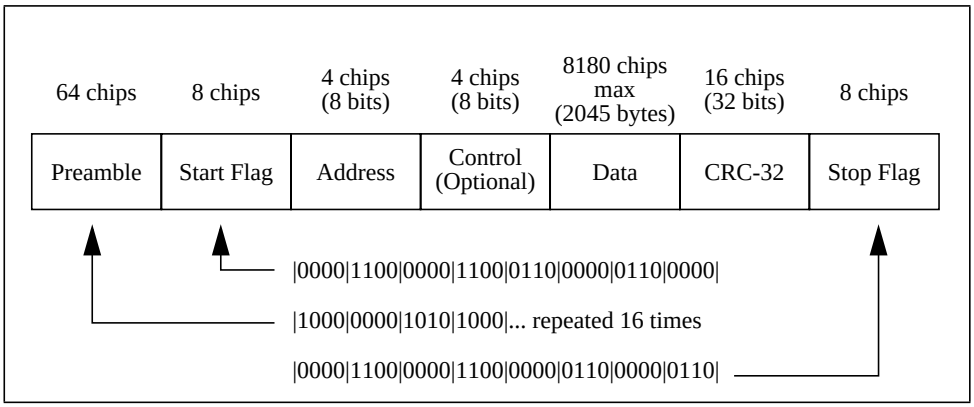


Figure 21. High Speed Serial Frame Format for IrDA Transmission (4.0Mbps)

The preamble, start, and stop flags are assembled from chips that contain either 0, 1, or 2 pulses. Chips with 0 and 2 pulses don't represent any valid encoding of (data) bit pairs and thus are used to distinguish flags from normal data.

The preamble contains 16 repeated transmissions of the four chips: 1000 0000 1010 1000; the start flag is a single sequence of eight chips: 0000 1100 0000 1100 0110 0000 0110 0000; the stop flag is similar to the start flag but the sequence of chips in the second half is rotated by one position: 0000 1100 0000 1100 0000 0110 0000 0110.

The other fields (address, control, data, and CRC-32) use the standard 4PPM encoding to represent two bits per chip.

5.3.2.1 Address Field

The 8-bit (4 chips) address field is used to either address a single station (addresses 0x00 to 0xFE) or to send a broadcast message to all stations within range (address 0xFF). The HSSP unit contains an eight bit register (AMV) which can be programmed to hold an address to be compared against the address field of incoming frames. If the address match function is enabled and the addresses match or if the incoming frame was sent to the broadcast address, the address is stored in the FIFO along with the normal data. The address is transmitted and received starting with its LSB and ending with its MSB.

5.3.2.2 Control Field

The 8-bit control field is optional and its meaning is defined by the user. There is no hardware support to handle it, the HSSP treats all bytes between the address field and the CRC as data and stores the bytes in the FIFO.

5.3.2.3 Data Field

In the standard the data field is defined to be any number of bytes from 0 to 2045. The length should be chosen according to the specific application and the transmission characteristics. The HSSP unit however does not limit the length of the data field in any way (except that it must be a multiple of eight bits, otherwise an abort is signalled).

5.3.2.4 CRC Field

Infrared high-speed mode uses a 32-bit cyclic redundancy check (CRC-32) to detect bit errors during transmission. The CRC is computed using the address, control, and the data field and is placed in the frame, between the data field and the stop flag. The HSSP contains separate CRC-generators for the transmit and the receive block. The transmitter calculates the CRC on the fly while the data is transmitted and places the inverse of the resulting 32-bit value at the end of each frame before the stop flag is transmitted. Similarly the receiver calculates the CRC for each received frame and compares the result to the received CRC field. If these two values do not match an interrupt is signalled. The CRC computation logic is preset to all ones before reception or transmission and the computed value is inverted before it is used for transmission or comparison. The 32-bit CRC is transmitted and received least significant byte first and most significant byte last. As usual within each byte the least significant nibble comes first.

The CRC uses the following 32-term polynomial:

$$CRC(x) = (x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1)$$

5.3.3 Baud Rate Generation

The infrared high-speed baudrate is 4 Mbps, thus a "chip frequency" of 2 MHz is required. Each chip consists of four time slots, resulting in a slot frequency of 8 MHz. Using a digital PLL the timeslot clock for the receiver unit is synchronized with the incoming data stream during preamble reception and afterwards whenever a transition is detected. The pattern of the preamble is used to identify the first time slot. As mentioned in Figure 20, the receive data are sampled with a frequency 6 times the slot frequency, capturing the sample value at the third clock of each slot period. The required sampling frequency of 48 MHz is taken from one of the two on-chip PLLs, the 8 MHz slot frequency is derived from this clock by division by six.

5.3.4 Receive Operation

Due to the nature of infrared communication the IrDA standard specifies that all communication has to be half-duplex, i.e., at any time a station can either transmit or receive, but not both. The HSSP hardware however does not impose such a restriction, transmitter and receiver can be enabled at the same time which is particularly useful when using the loopback mode to verify proper setup and operation of transmitter and receiver and the corresponding code..

When the receiver is enabled, it selects an arbitrary chip boundary, receives four incoming 4PPM chips from the RxD2 pin, and latches and decodes them one at a time. If the received chips do not decode to the four-chip preamble pattern, the time slot counter's clock is forced to skip one 8-MHz period, which delays the time-slot count by one. This procedure is repeated until the preamble pattern is recognized, signifying that the time-slot counter has synchronized. The four-chip preamble pattern is repeated at least 16 times, but can be transmitted continuously to signal an idle receive line.

At any time after the transmission of 16 preambles the start flag can be sent. This flag is eight chips long and if any part does not decode to the expected pattern the receiver logic signals a framing error and starts looking for the preamble pattern again.

After detection of a correct start flag each subsequent group of four chips is decoded into a data byte and placed in a 5-byte temporary FIFO. This FIFO is used to prevent the CRC from being placed within the receiver FIFO. When this temporary FIFO has been filled, the bytes are pushed out one by one and written to the receiver FIFO.

The first data byte of the frame is interpreted as the address. If receiver address matching is enabled the address byte is compared to the value stored in the address match register. If the two values are equal or the address byte is all ones, indicating a broadcast transmission, all following bytes, including the already received address byte are written to the FIFO. If the values do not match, nothing is stored in the FIFO and the receiver logic starts looking for the preamble again, ignoring any further data bytes. The optional control byte must be decoded in software if it is to be used.

The IrDA standard specifies a maximum of 2047 data bytes. There is no limit on the number of bytes that can be received by the HSSP receive logic, the user has to ensure that a received frame does not exceed the specified maximum frame length.

Similar to the UART FIFOs, when the receive FIFO is one- to two-thirds full, an interrupt request or DMA transfer is signalled. If the data is not read fast enough and the FIFO is completely filled, an overrun error is signalled, when the receiver logic tries to put additional data into the FIFO. Any subsequently received data bytes are discarded, but the contents of the FIFO remain intact.

If any two sequential chips within the data field do not contain any pulses (i.e., are 0000), the frame is aborted. The least recent byte within the temporary FIFO is moved to the receive FIFO, while the remaining four entries in the temporary FIFO are discarded. The end-of-frame (EOF) tag is set within the FIFO entry containing the last valid data byte. The receiver logic then begins to search for the next preamble. An abort also occurs if any received data chip contains 0011, 1010, 0101, or 1001 (invalid chips not contained in the stop flag).

During receive operation the receive logic continuously searches for the 8-chip stop flag. Once it is recognized, the last byte in the receive FIFO is marked as the last data byte of the frame. The bottom four entries of the temporary FIFO are compared to the continuously computed and updated CRC-32 value. If the received and the computed CRC value do not match the last byte in the receive FIFO is also tagged with a CRC error. The CRC value is not placed in the receive FIFO.

If the receiver is disabled during operation, reception of the current byte is stopped immediately, the FIFOs are cleared, all receiver-used clocks are stopped to save power and control of the RxD2 pin is given to the PPC unit.

Note: if the RxD2 pin is used for general purpose I/O care must be taken to ensure proper polarity of the pin.

5.3.5 Transmit Operation

Before enabling the HSSP transmitter the user can either fill the transmit FIFO using programmed I/O or wait for interrupt/DMA requests to fill the FIFO once the transmitter is enabled. For each frame to be transmitted at least 16 preambles are output. If there are no valid data in the FIFO after 16 preambles, the transmitter continues to output preambles until at least one byte of valid data resides in the transmit FIFO. The preambles are followed by the start flag and then the data (four chips, i.e., 8 bits are encoded at a time and then output on the TxD2 pin via a serial shifter clocked by the 8Mhz clock.

Note: preambles, start and stop flags, and CRC are automatically generated by the transmitter and need not be placed in the FIFO.

Whenever the FIFO is emptied at least halfway, a DMA request and/or an interrupt request are signalled. If the FIFO empties completely before new data is supplied, one of two actions (programmable by the user) can be taken - a FIFO underrun can either signal the regular completion of the frame or an unexpected termination of a frame being transmitted.

If the user has chosen normal frame completion and an underrun occurs, the transmitter sends the CRC which has been calculated during transmission of the frame (the CRC includes the address and control bytes). Following the CRC the stop flag is sent to signal the end of the frame. After that the transmitter again sends preambles until new data is available in the transmit FIFO in which case transmission of the new frame starts.

If “unexpected frame termination” has been programmed and a FIFO underrun occurs, the transmitter sends an abort and interrupts the CPU. The abort is transmitted until new data is available in the transmit FIFO. As soon as this happens, the transmitter starts transmission of the next frame by sending the 16 preambles, followed by start flag and data.

At the end of each frame, the HSSP outputs a pulse called the serial infrared interaction pulse (SIP). A SIP is required at least every 500ms to keep SIR-devices (speeds up to a maximum of 115kBaud) from interfering with the FIR transmission. The pulse simulates a start bit which causes all SIR-devices to await their turn for transmission for at least another 500ms. The SIP pulse is generated by setting the transmit pin (TxD2) high for a period of 1.625µs, followed by a low period of 7.375µs (total length 9µs). After this 9µs period normal operation is resumed by transmitting preambles until valid data is available in the transmit FIFO. It is the user’s responsibility that a SIP pulse can be generated at least every 500ms. As most IrDA compatible devices produce a SIP pulse after each transmitted frame it is enough to ensure that at least one frame is either transmitted or received every 500ms.

Note that this is no limitation regarding the frame length - the maximum frame length is 16376 bits which takes 4ms at 4Mb/s.

If the transmitter is disabled during operation, transmission of the current byte is immediately stopped, the serial shifter and the transmit FIFO are cleared, and control of the TxD2 pin is given to the PPC. Also all clocks needed for transmit operation are shut down to save energy. Again, if the pin is to be used for general purpose I/O care must be taken to ensure proper polarity of the pin.

5.3.6 Transmit and Receive FIFOs

As in the case of the UART FIFOs, the HSSP’s FIFOs use self-timed logic to reduce power consumption and chip size. As explained in Section 4.8.3, the depth at which a service request to empty the **receive** FIFO is triggered, is variable. The variability can span up to four FIFO entries. To compensate for this variability and to allow efficient use of the high datarate, the receiver FIFO has 20 entries. The receive FIFO generates a service request if it is filled from two-fifths to three-fifths (nine to twelve entries of data).

The transmit FIFO has 16 entries and is guaranteed to signal a service request if there are eight or more free entries and to negate the request if there are less than eight free entries.

For DMA operation the burst size must be set to eight words. With programmed I/O eight bytes can be written to the transmit FIFO or read from the receive FIFO without checking, following that the appropriate status bits must be polled.

5.3.7 HSSP Register Description

The HSSP unit contains six registers, three control registers, one data register and two status registers. The HSSP memory map is shown in Table 18, the base address is 0x80040000.

Offset	Name	Description	Access
60	HSCR0	HSSP control register 0	R/W
64	HSCR1	HSSP control register 1	R/W
6C	HSDR2	HSSP data register	RW
74	HSSR0	HSSP status register 0	R/W & RO
78	HSSR1	HSSP status register 1	RO
0x9006 0028	HSCR2	HSSP Control Register 2	R/W

Table 18. HSSP register block

Note: HSSP Control Register 2 is located in the **PPC address space** at 0x9006 0028.

5.3.7.1 HSSP Control Register 0 (HSCR0)

Bit	Name	Description
0	ITR	Irda transmission rate. 0 - 115.2Kbps (select HP-SIR modulation, enable serial port 2 UART) 1 - 4.0 Mbps (select 4PPM modulation, enable serial port 2 HSSP) Note: this is the only bit in this register that affects both the UART and the HSSP; once one of the two modes is selected, all further programming is controlled by the individual units (UART or HSSP).
1	LBM	Loopback mode. 0 - Normal serial port operation enabled. 1 - Output of HSSP's transmit serial shifter is directly connected to the input of the receive serial shifter. Control of TxD2 and RxD2 is given to the PPC unit if ITR=1. Note: the IrDA standard requires half-duplex operation but the HSSP's hardware allows full-duplex operation which is essential for the loopback mode.
2	TUS	Transmit FIFO underrun select. 0 - Transmit FIFO underrun masks transmit underrun interrupt generation (status flag TUR in HSSP status register 0 is ignored) and causes frame to be finished regularly (transmitting CRC, stop flag, and SIP). 1 - Transmit FIFO underrun causes an abort to be transmitted, and generates an interrupt (state of TUR is evaluated by the interrupt controller).
3	TXE	Transmit enable. 0 - HSSP transmit logic is disabled; control of the TxD2 pin is given to the PPC unit if ITR=1 1 - HSSP transmit logic is enabled if ITR=1. Note: immediately after enabling the transmitter a SIP pulse is transmitted.
4	RXE	Receive enable. 0 - HSSP receive logic is disabled; control of the RxD2 pin is given to the PPC unit if ITR=1. 1 - HSSP receive logic is enabled if ITR=1.
5	RIE	Receive FIFO interrupt enable. 0 - "Receive FIFO two- to -three-fifths full or more" condition does not generate an interrupt (RFS bit ignored). 1 - An interrupt is generated if the receive FIFO is two- to three-fifths full or more.

Table 19. HSSP Control Register

Bit	Name	Description
6	TIE	Transmit FIFO interrupt enable. 0 - “Transmit FIFO half-full or less” condition does not generate an interrupt (TFS bit ignored). 1 - An interrupt is generated if the transmit FIFO is half-full or less.
7	AME	Address match enable. 0 - Disable receiver address match function; store data from all incoming frames in the receive FIFO. 1 - Enable receiver address match function; only data from frames with matching address or broadcast address (all ones) is stored in the receive FIFO.

Table 19. HSSP Control Register

Some more remarks:

- Transmit FIFO Underrun Select (TUS): when TUS=0, transmit FIFO underruns signal to the transmit logic that the end of the frame has been reached and the frame is to be terminated with CRC, stop flag, and SIP pulse. Additionally the transmit FIFO underrun interrupt request is masked.

  When TUS=1 a transmit FIFO underrun signals that the end of the frame has not yet been reached but the data rate to fill the FIFO was not sufficient. The transmitter outputs two chips containing all zeros (0000) to signal the abort. This is followed by a SIP pulse and a minimum of 16 preambles. Transmission restarts as soon as new data is available within the transmit FIFO. Additionally setting TUS=1 enables the transmit FIFO underrun interrupt.

  For normal operation TUS should be set to 1 at the start of a frame in order to generate aborts if the FIFO underruns. Just before the end of the frame the user clears TUS to 0. When the FIFO underruns the frame is properly terminated. Before changing the state of TUS the FIFO should be filled in order to avoid the FIFO to underrun at the same time TUS is written.
- Transmit Enable (TXE): enable/disable HSSP transmit operation. When TXE=0 the transmit logic is disabled, its clocks are turned off in order to save power, and control of the TxD2 pin is given to the PPC unit. As with the UARTs it is required that all other bits be programmed before setting TXE to 1. TXE is ignored if ITR=0 (UART mode), TXE and RXE are the only bits in HSSP control register 0 that are reset to a known state (0).
- The Address Match Enable (AME) bit is used to enable or disable the receive logic form comparing the address programmed in the Address Match Value (AMV) field to the address of the incoming frames.

  If AME=1 data of received frames is stored in the receive FIFO only if the value in AMV and the address field of the incoming frame match or if the incoming frame was sent with the broadcast address (all ones). For frames whose address does not match, data and CRC are ignored and the receiver logic keeps looking for the next preamble.

  If AME=0, the address values are not compared and the data in every frame is stored in the receive FIFO.

5.3.7.2 HSSP Control Register 1 (HSCR1)

HSSP control register 1 (HSCR1) is used to store the 8-bit address match value AMV.

The address match mechanism can be used to filter out frames, that are not destined for the local station, by means of hardware, without using processor time. To enable this function the desired address match value has to be written to HSCR1 and the AME bit must be set. Then for matching incoming frames the frame’s address, control and data are stored in the receive FIFO. Frames, whose address does not match, are discarded and the receive logic searches for the next preamble to synchronize on. Frames containing the broadcast address 0xFF always match, their contents are automatically stored in the receive FIFO.

AMV is uninitialized after reset and can be written at any time, the update becomes active with the next frame.

5.3.7.3 HSSP Control Register 2 (HSCR2)

Unlike all other HSSP registers this register is located in the PPC address space. It contains two read- and writable bits that are located in byte 2 of the addressed word (bits 23..16), word writes or reads should be used to access this

register. These two bits determine the polarity of the TXD2 and RXD2 pins. Both bits are reset to 1 resulting in “true” mode for both pins.

- Transmit Pin Polarity Select (TXP): This bit is located in bit 18 of HSCR2 and determines whether data output via the TXD2 pin is output true or complemented. When TXP=0, all data output via this pin (UART, HSSP, and PPC) first is inverted. When TXP=1 all data is output true/non inverted. This applies only for output, if the pin is used for general purpose I/O input (PPC), TXP has no effect on the state of TXD2.

  Note: if TXP=0, indicating inverted data, also the corresponding bit in the PPC sleep state register has to be inverted.
- Receive Pin Polarity Select (RXP): RXP is located in bit 19 of HSCR2 and controls the behaviour of the serial port 2 receive pin RxD2. If RXP=0, data input from the RxD2 pin is first inverted before being sent to the UART, the HSSP or the PPC. When RXP=1, data input from the RxD2 pin is not inverted before being sent to its destination unit. The bit is also set to 1 after reset, but unlike the TXP bit it controls the PPC GPIO **input** on pin RxD2. Also unlike TXP, it has no effect on the PPC sleep state register, therefore the corresponding bit PSDR<15> should be programmed normally.

5.3.7.4 HSSP Data Register (HSDR)

The HSSP Data Register (HSDR) is an eight-bit register corresponding to both the top and the bottom entry of the transmit and receive FIFOs, respectively.

The mechanism is the same as with the UART FIFOs: when HSDR is read the lower eight bits of the bottom entry of the 11-bit wide receive FIFO are accessed. As data enters the top of the receive FIFO, bits 8-10 are used as tags to indicate various conditions related to each received data byte. The tags are transferred down the FIFO along with the data byte they belong to. When the data byte reaches the bottom position of the FIFO the tag bits are transferred to the status bits end-of-frame (EOF) flag, CRC error (CRE) flag, and the receiver overrun (ROR) flag, all in status register 1. These bits can be checked by the user to detect the end of a frame or an error condition before reading the corresponding data byte.

Again in analogy to the UART FIFOs the ‘end/error in FIFO’ (EIF) flag is set in status register 0, if any of the tag bits is set within the eight bottom entries of the receive FIFO and is cleared if no tag bit is set in the bottom eight entries. If EIF is set, DMA is disabled and an interrupt is generated. The user then can check which condition lead to the interrupt by checking the status flags in status register 1 and read the data bytes one at a time until there are no more data bytes with set tag bits left within the bottom eight receive FIFO entries. At this point the EIF flag is cleared automatically and DMA is reenabled.

When HDSR is written, the topmost entry of the 8-bit transmit FIFO is accessed. Data is transferred to the lowest location within the FIFO that does not yet contain valid data and transmission starts as soon as a byte reaches the bottom and the transmitter is enabled.

5.3.7.5 HSSP Status Register (HSSR0)

HSSP status register 0 is an eight bit wide register that contains six bits to signal FIFO service requests and error conditions. Each of the bits signals an interrupt request. Read/write bits are called status bits and have to be cleared by the user by writing a one to the bit, read-only bits are called flags and are cleared by hardware, they are not

affected by any write operation. The reset state of all writable bits is unknown and has to be cleared before enabling the HSSP.

Bit	Name	Description	Access
0	EIF	End/Error in FIFO. 0 - Bits 8-10 are not set within any of the eight bottom entries of the receive FIFO. 1 - At least one of the tag bits is set within the bottom eight entries of the receive FIFO.	RO
1	TUR	Transmit FIFO underrun. 0 - Transmit FIFO has not experienced an underrun. 1- Transmit logic attempted to fetch data from the transmit FIFO while it was empty. This generates an interrupt request, if TUS=1.	R/W
2	RAB	Receiver Abort. 0 - No abort has been detected for the incoming frame. 1 - During receipt of the incoming frame an abort (two or more chips containing no pulses) has been detected. The EIF bit is set in the receive FIFO next to the last valid data byte and an interrupt service is requested.	R/W
3	TFS	Transmit FIFO service request. 0 - Transmit FIFO is more than half-full (nine or more entries filled) or the transmitter is disabled. 1 - Transmit FIFO is half-full or less (eight or fewer entries are contained). DMA service request is signalled and also an interrupt service request, if TIE=1. Note that the DMA request is not affected by the state of RIE. After CPU or DMA have written new data to the FIFO, and it thus contains eight or more valid data bytes, TFS is automatically cleared.	RO
4	RFS	Receive FIFO service request. 0 - Receive FIFO contains 11 or fewer entries of data or the receiver is disabled. 1 - Receive FIFO is two- to three-fifths full (9-12 entries) or more. DMA and interrupt service request are signaled, the latter only if RIE=1. Note that the DMA request is not affected by the state of RIE.	RO
5	FRE	Framing Error. 0 - no framing errors encountered 1 - a framing error has occurred (a preamble followed by something other than another preamble or start flag.	R/W
7..6	-	Reserved.	

Table 20. HSSP Status Register 0 (HSSR0)

Some more remarks on the bits in status register 0:

- End/Error in FIFO Flag (EIF): the bit is set, if any of the eight bottom entries in the receive FIFO contains a set tag bit. (Receiver) DMA is disabled and an interrupt request is signalled. The user should then check the state of the bits EOF, CRE, and ROR in HSSP status register 1 and read the next data byte in the receive FIFO using programmed I/O. This has to be repeated until none of the tags is set in the bottom eight entries of the FIFO any more which automatically clears the EIF flag and reenables DMA.
- Transmit Underrun Status (TUR): The bit is set if the transmit logic tries to fetch data from the transmit FIFO after this has already been emptied. Further action depends on the state of the TUS flag in control register 0:
  - TUS=0: the transmitter ends the frame regularly by transmitting the accumulated CRC value, followed by a stop flag and a SIP pulse.
  - TUS=1: the transmitter sends abort chips (containing all zeros) until new data is available in the transmit FIFO. Then a new frame is initiated by sending preambles and the start flag, followed by the newly available data. If TUS=1an interrupt request is signalled to notify the CPU/user program that the data rate is too low.
- Receiver Abort Status (RAB): This bit is set when an abort is detected during receipt of an incoming frame. The abort condition is that two or more chips that do not contain any pulses (all zeros) or chips containing 0011, 1001, or 0101 (i.e., chips that do not represent any valid data encoding and are not contained in the stop flag) are received after a valid start flag but before a valid stop flag has been detected. When an abort condition occurs the

EOF tag is set in the receiver FIFO entry that holds the last valid data byte. The receiver then starts searching for a preamble.

- Receive FIFO Service Request Flag (RFS): set to indicate that the receive FIFO requires service to prevent an overrun. As with the UART FIFOs the service request is guaranteed to occur only in a certain range, not with a certain number of bytes contained in the FIFO. In the case of the HSSP receive FIFO it is guaranteed to signal a service request when it is two- to three-fifths full (or more), i.e., when it contains at least 9 to 12 valid entries. The request is cleared when the FIFO contains 9 to 11 remaining entries. The DMA burst size must be set to eight words, using (interrupt driven) programmed I/O up to eight bytes can be read without checking, then the RNE flag has to be checked if additional data is available.

5.3.7.6 HSSP Status Register 1 (HSSR1)

Status register 1 contains seven non-interruptible read-only bits to indicate receiver and transmitter state, FIFO state, and frame state info.

Bit	Name	Description
0	RSY	Receiver synchronized flag. 0 - Receiver is in hunt mode or is disabled. 1 - Receiver logic is synchronized with the incoming data.
1	TBY	Transmitter busy flag. 0 - Transmitter is idle (sending continuous preambles) or is disabled. 1 - Transit logic is currently transmitting a frame (address, control, data, CRC, or start/stop flag).
2	RNE	Receive FIFO not empty flag. 0 - Receive FIFO is empty. 1 - Receive FIFO contains at least one valid entry.
3	TNF	Transmit FIFO not full flag. 0 - Transmit FIFO is full. 1 - Transmit FIFO has space for at least one more data byte.
4	EOF	End of frame flag. 0 - Current frame has not yet completed. 1 - The value at the bottom of the receive FIFO is the last byte of data within the current frame.
5	CRE	CRC error flag. 0 - No CRC errors encountered in the receipt of data. 1 - CRC calculated on the incoming data does not match the CRC value contained within the current frame.
6	ROR	Receive FIFO overrun flag. 0 - Receive FIFO has not experienced an overrun. 1 - Receive logic attempted to place data into the receive FIFO while it was full. The next data byte to be read is the last “good” one before the overrun occurred.

Table 21. HSSP Status Register 1 (HSSR1)

Some more remarks:

- Receive FIFO Not Empty Flag (RNE): the bit is set whenever the receive FIFO contains at least one byte of valid data and is cleared when the FIFO is empty. It can be polled when using programmed I/O to remove any remaining bytes after a DMA transfer or an interrupt driven access (which can remove only eight bytes at a time although more bytes may be contained).
- Transmit FIFO Not Full Flag (TNF): the flag is set whenever the transmit FIFO contains at least one empty entry and is cleared when the transmit FIFO is completely full. Similar to the RNE flag, this flag can be polled when using programmed I/O to fill the transmit FIFO over the halfway mark.

The following three flags represent the tag bits (bits 8-10 of the receive FIFO that travel along with the corresponding data byte) of the current bottom entry of the receive FIFO. When one of the tag bits is set within the bottom eight FIFO entries the EIF flag is set and generates an interrupt request. As long as this bit is set, the user should check the three status register 1 flag bits described in the following before reading the next byte out of the FIFO.

- End-of-Frame Flag (EOF): bit 8 in the receive FIFO is set when the last byte of data within a frame (including aborted frames) is moved from the receive serial shifter to the top of the receive FIFO. When this FIFO entry reaches the bottom of the FIFO, its bit 8 is moved to the EOF flag bit to indicate that the last byte of data of the current frame resides within the bottom entry of the receive FIFO.
- CRC Error Status (CRE): The CRC Error Status flag is moved from bit 9 of the receive FIFO when the corresponding FIFO entry reaches the bottom position. It indicates that the new bottom entry in the FIFO is the last byte of a frame whose transmitted checksum value did not match the computed checksum.
- Receiver Overrun Status (ROR): this bit is moved from bit 10 in the receive FIFO when the corresponding data entry reaches the bottom position and indicates that the new bottom entry is the last valid data byte before the receiver overrun occurred.

5.4 Support Code for Infrared Communication

The code for infrared communication on the SmartBadge is split into three files, a common header file declaring all the constants, macros and function prototypes, and two C-files, containing the code for SIR mode and FIR mode, respectively. By defining or undefining the symbol SIR_UART_TRANSMISSION the SIR code can be configured to use software modulation or normal UART functionality (for use on revised versions of the SA-1100). As this file has to be configured and recompiled for each application it should be added to the project as another source file while the file containing the FIR code can be added to the library containing the support code for the other peripherals. The support code can be found in Appendix A ”Infrared support code”, some basic example programs demonstrating the use of this support code are listed in Appendix B.6 ”IR SIR-Mode Examples” and Appendix B.7 ”IR FIR-Mode Examples”.

5.5 Setup to Debug Infrared Communication

During my work on the infrared code I frequently had to watch the infrared communication in order to be able to determine the source of errors. Especially in the early stages of the implementation of the IrDA protocol stack the ability to actually watch what was transmitted was a necessity for making progress. By listening to the communication between an infrared access point and a notebook I could acquire important information about the protocols. It was also necessary to check my own transmission routines, especially at the stage where I was working with the software modulation.

Apart from using the logic analyzer at a very low level at the beginning to watch and check the transmission of single bits I used two methods:

- An HP Omnibook 600 which is equipped with a UART compatible infrared port. By configuring the infrared port as a serial port I could directly record the characters being transmitted via the infrared link.
- My second method was to use another SmartBadge and use its infrared port for listening. During my work on the IrDA stack I used the second Badge to just record the sent characters and then on demand transmit them to a PC running Linux where I had a small program that analyzed the received data to check it for the frame delimiters and the CRC. Using a second SmartBadge for recording allowed an orientation such that one Badge was talking to the access point whereas the other SmartBadge could receive information from both the access point and the first SmartBadge at the same time. Using only a notebook this would not have been possible as the notebooks usually have a lens in front of the infrared transceiver that limits the receiving angle. As the SmartBadges are not packaged they have a bigger angle and probably also reflections were helping quite a lot in this case.

6. Accessing Peripherals, Code Generation with the ARM Compiler

As described in the overview of the StrongARM and the chapter about the peripherals, peripherals are accessed via memory mapped registers which are manipulated by the load and store instructions provided in the ARM architecture. In the type of applications potentially using the StrongARM processor I/O-operations represent a considerable fraction of the whole code. Therefore it's worth trying to minimize that code both in terms of codesize and execution time. As Tatjana Simunic points out in her paper [10] in regard to the power consumption even the specific type of memory used - SRAM, DRAM (available only on SmartBadge Version 4), or FLASH - is of importance. This section deals with some issues concerning the code generated by the compiler in the ARM Software Development Tools v2.1. In the first part I describe a few different ways of accessing the registers that control the peripherals. In the following section I examine the assembly code produced by the compiler for each of the different coding methods using the different optimization levels provided by the compiler and compare the results in terms of instructions and memory consumption.

6.1 Methods of Accessing the Peripherals' Registers

As an example I use a simple initialization function for UART 1 and describe four ways of accessing the necessary registers. Although the initialization might not be executed frequently, it serves as a nice example as it shows the basic instructions which are necessary for every interaction with a device.

If the content of a memory location can be changed by different tasks, e.g. it is accessed in a normal function but also in an interrupt routine or, as in the case of memory mapped registers, can be modified directly by hardware, problems can arise if the compiler does not know about this fact and tries to optimize the code. The problems usually occur if the variable is accessed multiple times, for instance in a polling loop in which case the compiler might move the code to access the variable out of the loop. To avoid this the variable used to access the memory location must be declared as `volatile`. In the case of the initialization function described in the next sections, all registers are accessed only once, therefore the problem does not occur. To demonstrate the problem I give an example at the end of this chapter where the compiler generates incorrect code if the variable is not declared as `volatile`.

6.1.1 Declaration as const unsigned int

For each register a variable of type `const unsigned int` is declared and initialized with the address of that register. To read or write the register the variable is cast to a pointer and then read/assigned the new value.

```
// UART1 Port addresses
const unsigned int const_UT1CR0 = 0x80010000; //UART1 Control Register 0
const unsigned int const_UT1CR1 = 0x80010004; //UART1 Control Register 1
const unsigned int const_UT1CR2 = 0x80010008; //UART1 Control Register 2
const unsigned int const_UT1CR3 = 0x8001000C; //UART1 Control Register 3

// set up UART1 for 9600 baud with everything enabled to go
void const_init(void)
{
    // control register 0 (basic setup)
    *(volatile unsigned int *)const_UT1CR0 = 0x00000008;

    // control register 1 & 2 (baud rate)
    *(volatile unsigned int *)const_UT1CR1 = 0x00000000;
    *(volatile unsigned int *)const_UT1CR2 = 0x00000001;
```

Listing 14. Register access with `const unsigned int`

```
    // control register 3 (transmitter and receiver enable)
    *(volatile unsigned int *)const_UT1CR3 = 0x00000003;
}
```

Listing 14. Register access with `const unsigned int`

6.1.2 Declaration as Pointer

For each register a pointer of type `volatile unsigned int*` is declared and then dereferenced to read or write the register.

```
// UART1 Port addresses pointer declaration
volatile unsigned int* pUT1CR0 = (unsigned int*)0x80010000; //UART1 Control Register 0
volatile unsigned int* pUT1CR1 = (unsigned int*)0x80010004; //UART1 Control Register 1
volatile unsigned int* pUT1CR2 = (unsigned int*)0x80010008; //UART1 Control Register 2
volatile unsigned int* pUT1CR3 = (unsigned int*)0x8001000C; //UART1 Control Register 3

// set up UART1 for 9600 baud with everything enabled to go
void pointer_init(void)
{
    // control register 0 (basic setup)
    *pUT1CR0 = 0x00000008;

    // control register 1 & 2 (baud rate)
    *pUT1CR1 = 0x00000000;
    *pUT1CR2 = 0x00000001;

    // control register 3 (transmitter and receiver enable)
    *pUT1CR3 = 0x00000003;
}
```

Listing 15. Register access with pointers

6.1.3 Declaration as Structure

First a structure describing the register block is defined. To access the registers a pointer of that type is declared and initialized with the base address of the register block. Then each register can be accessed via the corresponding entry in the structure.

```
/* structure describing the UARTs on the Badge */
typedef volatile struct
{
    unsigned int UTCR0;
    unsigned int UTCR1;
    unsigned int UTCR2;
    unsigned int UTCR3;
    unsigned int UTCR4;
    unsigned int UTDR;
    unsigned int reserved;
    unsigned int UTSR0;
    unsigned int UTSR1;
} uart;

// UART1 port address pointer declaration
uart* pUART = (uart*) 0x80010000;  //UART1 Base Address

// set up UART1 for 9600 baud with everything enabled to go
void struct_init(void)
{
    // control register 0 (basic setup)
    pUART->UTCR0 = 0x00000008;

    // control register 1 & 2 (baud rate)
```

Listing 16. Register access using a structure

```
    pUART->UTCR1 = 0x00000000;
    pUART->UTCR2 = 0x00000001;

    //  control register 3 (transmitter and receiver enable)
    pUART->UTCR3 = 0x00000003;
}
```

Listing 16. Register access using a structure

6.1.4 Use of #define

For each peripheral a `#define` is used to specify the base address of that register block. Then there is another `#define` for each of the registers, giving the offset of that register to the base address.

To access a register, its address (adding base address and offset) is cast to a pointer of type `volatile unsigned int`. This can be conveniently done in a preprocessor macro.

```
/* base address of UART 1 */
#define UART1_BASE 0x80010000

/* UART regs/values offsets from base address */
#define UTCR0 0x00    // UART Control Register 0
#define UTCR1 0x04    // UART Control Register 1
#define UTCR2 0x08    // UART Control Register 2
#define UTCR3 0x0C    // UART Control Register 3

/* access a register by its base address and an offset */
#define REG(base, offs) (*(volatile unsigned int*)(base+offs))

// set up UART1 for 9600 baud with everything enabled to go
void define_init(void)
{
    // control register 0 (basic setup)
    REG(UART1_BASE, UTCR0) = 0x00000008;

    // control register 1 & 2 (baud rate)
    REG(UART1_BASE, UTCR1) = 0x00000000;
    REG(UART1_BASE, UTCR2) = 0x00000001;

    // control register 3 (transmitter and receiver enable)
    REG(UART1_BASE, UTCR3) = 0x00000003;
}
```

Listing 17. Register access using #define

Note: Preprocessor macros and definitions can be used to enhance this declaration style in order to generalize accessing any devices. New macros can be defined for the various actions, like `device_read_status(device, register)`, `device_set_control(device, register, value)`, `device_read_data(device, len, buf)`, `device_write_data(device, len, buf)`. Based on the parameter `device` these macros can be simple renamings of the basic macro `reg(base, offset)`, or calls to specialized functions if the action cannot be done in a simple macro (e.g. to support devices that are to be accessed interrupt-driven or via DMA on Badge4, serial port 4,...). The important point is, that all of this can be done at compile time. It is thus possible to have a generic interface to all devices without runtime overhead like for instance function tables which are widely used for this purpose.

6.2 Code Produced by the Compiler

In the ARM Project Manager projects are subdivided into so called “variants”. All the variants have the same set of source files, but they can have different options. The usual variants are “Debug” and “Release”. These two are already predefined when a new project is created. There are three sets of compiler options which influence the code generation:

- Debug Control
  - Enable debug table generation
  - Include preprocessor symbols
- Source Level Debug Optimization
  - None
  - No register allocation optimization
  - Optimize fully
- Optimize Code
  - Default balance
  - For space
  - For time

These can be set on the tab folder “C & Debug” (Project->Tool Configuration for ...-><cc>=armcc->set in the ARM Project Manager) or by the command line options.

The primary difference between the “Debug” and the “Release” variant is that in the “Debug” variant the option “Enable debug table generation” is enabled. It turns out that for the chosen example when this option is enabled, the option group “Source Level Debug Optimization” influences the resulting code, while the options in the group ”Optimize Code” have no influence. If the option “Enable debug table generation” is disabled, the resultant code is not influenced by any of the options in either of the two groups - the ARM compiler always creates “its” optimal code (which is not in all cases the theoretically optimal code). Therefore I will only discuss the three cases for “Source Level Debug Optimization” with “Enable debug table generation” enabled. The third case - “Optimize fully” - generates the same code as all cases with “Enable debug table generation” disabled. Of course with other code there will be differences between the different optimization levels. However, in any case the issues I am talking about in the next sections remain valid.

In the following sections I will first give a short introduction to the load/store instructions available in the ARM instruction set. A more detailed description can be found in the ARM Architectural Reference Manual ([5]). Then I will compare the generated code for the three optimization levels using the different coding variants. It turns out that variants one (see section 6.1.1) and two (see section 6.1.2) are just notational variants, the code produced is exactly the same, so I will describe these two cases together. There is a difference though in memory allocation - the **`const`** declarations are stored in a read only data segment (FLASH) while the **pointer** declarations are put into a read-write data segment, i.e. in the (S)RAM. As mentioned in the introduction to this chapter this difference may be significant with respect to speed and power consumption.

6.2.1 ARM load/store Instructions

The ARM architecture provides two different ways to load a register with a value:

- The move instruction
- the load instructions

The move instruction is used to move a value from one register to another, to put a constant value into a register and to perform a shift without any other arithmetic or logical operation. The general format is as follows:

```
MOV{<cond>} Rd, <shifter_operand>
```

MOV moves the value of `<shifter_operand>` into `Rd` and, based on the result, updates the condition code flags. Like all other ARM instructions it is only executed if the condition <cond> (basically a code for the processor status flags) is true. If Rd is the program counter register (PC) a branch occurs.

There are eleven types of <shifter_operand>, including immediate value rotated right, registers - directly or logically/arithmetically shifted left/right by an immediate value in the range 0 to 31 or by the value in a second register, or rotated right by an immediate value or a value in a second register.

The immediate format produces a value by rotating an 8-bit constant value to the right by an even number of bits. That implies that not all 32-bit constants can be produced using this instruction. Unavailable values must be obtained by combining the MOV instruction with arithmetic operations or by loading a value from a memory location. This is accomplished via the load instruction LDR (the counterpart for storing values from a register to memory (STR) uses the same format):

```
LDR{<cond>} Rd, <addressing mode>
```

<cond>	the instruction is only executed if the condition is passed
Rd	the destination register for the load operation
<addressing mode>	The ARM instruction set offers nine addressing modes used to calculate the address for store/load operations of words or unsigned bytes. They can be divided into three groups:
“Normal mode”	the value at address is loaded into Rd
pre-indexed mode	the offset is applied to the base register, the result is used as the address of the location to load into Rd as in normal mode, but additionally, if <cond> is passed, the base register is updated with the new address.
post-indexed mode	the base register gives the value used for addressing the storage location. Afterwards, if <cond> was passed, the offset is applied to the base register and written back.

Each of the three modes supports the following three types of instruction:

Immediate offset	`Rn, #+/-<12_bit_offset>`	load the value at location Rn+/-offset into register Rd.
Register offset	`Rn, +/-Rm`	load the value at location Rn+/-Rm into register Rd.
Scaled register offset	`Rn, +/-Rm, <shift>#<shift_imm>`	load the value at location Rn+/- (shifted/ rotated value of Rm) into register Rd.

Note that instructions of the format `LDR Rd, offset` are pseudo-instructions - actually they are encoded as `LDR Rd, pc, #offset` to give an offset relative to the current location.

6.2.2 Code Examples for the Initialization Function with no Optimizations Enabled

Without optimizations enabled the compiler generates the following code for the `const`/pointer variant:

```
20           // set up UART1 for 9600 baud with everything enabled to go
21           void const_init(void)
22           {
23              // control register 0 (basic setup)
24              *(unsigned int *)const_UT1CR0 = 0x00000008;
   0x00008194   mov      r1,#8
   0x00008198   ldr      r0,0x000081d8 ; = #const_UT1CR0
   0x0000819c   ldr      r0,[r0,#0]
   0x000081a0   str      r1,[r0,#0]
26
27              // control register 1 & 2 (baud rate)
28              *(unsigned int *)const_UT1CR1 = 0x00000000;
   0x000081a4   mov      r1,#0
   0x000081a8   ldr      r0,0x000081d8 ; = #const_UT1CR0
   0x000081ac   ldr      r0,[r0,#4]
   0x000081b0   str      r1,[r0,#0]
29              *(unsigned int *)const_UT1CR2 = 0x00000001;
   0x000081b4   mov      r1,#1
   0x000081b8   ldr      r0,0x000081d8 ; = #const_UT1CR0
   0x000081bc   ldr      r0,[r0,#8]
   0x000081c0   str      r1,[r0,#0]
30
31              // control register 3 (transmitter and receiver enable)
32              *(unsigned int *)const_UT1CR3 = 0x00000003;
   0x000081c4   mov      r0,#3
   0x000081c8   ldr      r1,0x000081d8 ; = #const_UT1CR0
   0x000081cc   ldr      r1,[r1,#0xc]
   0x000081d0   str      r0,[r1,#0]
33           }
   0x000081d4   mov      pc,r14
   0x000081d8   dcd      0x0000aa28
```

Listing 18. Code for const declaration without optimisations

It turns out that the compiler generates four assembly instructions for each of the four assignments. The first four instructions are necessary, but for the following three assignments the code could be reduced as the base address remains the same.

The code of the first two variants results in a memory layout like that shown in Figure 22:

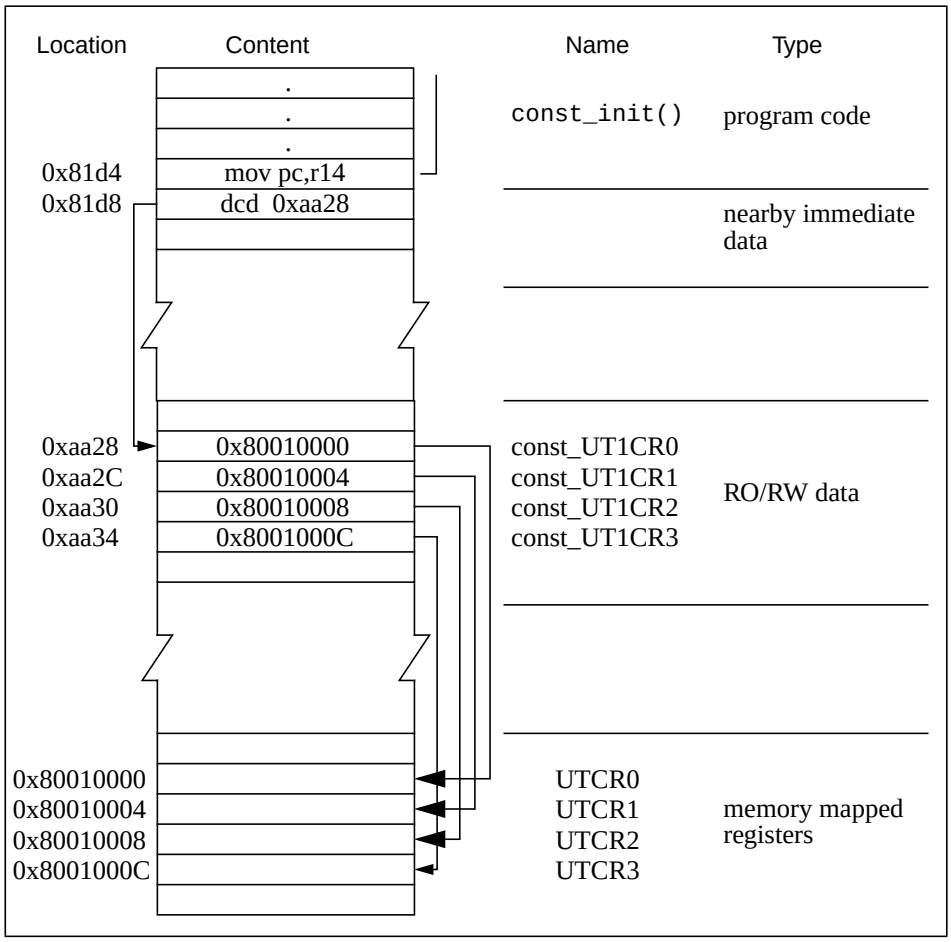


Figure 22. Memory layout for const/pointer variant

The location of the peripheral registers is fixed, the location of the four variables depends on their declaration - read only memory for the constant declaration, RAM for the pointer declaration. To write a value (#8) to one of the peripheral registers first this value is loaded into a processor register using the move instruction:

```
0x00008194  mov       r1,#8
```

Then the content of the variable const_UT1CR0 which holds the address of the register to access must be loaded. The address of this variable is a 32-bit immediate value which cannot be generated by the move instruction. Therefore the value is stored in a location immediately after the end of the function. This location is accessible using an immediate offset:

```
0x00008198  ldr       r0,0x000081d8 ; = #const_UT1CR0
```

This output has been transformed to be easier to read - in fact the instruction is encoded as:

```
ldr r0, pc, +#offset
```

where `#offset = 0x81d8 - address(current instruction) - 8`. The subtraction of eight is the result of the pipelined architecture.

Now register r0 holds the value 0xaa28 which is the address of the variable. The content of the variable (i.e. the address of the register to access) can now be loaded:

```
0x0000819c  ldr       r0,[r0,#0]
```

Now r0 holds the memory address of the peripheral register to access and the new value can be stored there:

```
0x000081a0  str      r1,[r0,#0]
```

This code sequence is repeated for each of the three following assignments. However, as I have mentioned before there is no difference in the generated code between the pointer and the `const` variant in any of the three optimization levels. This shows that the optimization methods in the compiler are not very effective. As the pointer values could be changed in the pointer case it is hard to tell that the targeted memory locations all lie within a small offset and therefore the code could have been optimized. But in the constant case the compiler could analyse the contents of the variables and do optimizations. An interesting detail is that in the first three assignments register r1 is used to hold the new value while register r0 is used to hold the address. In the fourth assignment this order is swapped.

When using a pointer to a structure describing the register block the compiler produces the following code:

```
20          // UART1 port address pointer declaration
21          uart* pUART = (uart*) 0x80010000;  //UART1 Base Address
22
23          // set up UART1 for 9600 baud with everything enabled to go
24          void struct_init(void)
25          {
26             // control register 0 (basic setup)
27             pUART->UTCR0 = 0x00000008;
   0x000080cc   mov      r1,#8
   0x000080d0   ldr      r0,0x00008110 ; = #pUART
   0x000080d4   ldr      r0,[r0,#0]
   0x000080d8   str      r1,[r0,#0]
28
29             // control register 1 & 2 (baud rate)
30             pUART->UTCR1 = 0x00000000;
   0x000080dc   mov      r1,#0
   0x000080e0   ldr      r0,0x00008110 ; = #pUART
   0x000080e4   ldr      r0,[r0,#0]
   0x000080e8   str      r1,[r0,#4]
31             pUART->UTCR2 = 0x00000001;
   0x000080ec   mov      r1,#1
   0x000080f0   ldr      r0,0x00008110 ; = #pUART
   0x000080f4   ldr      r0,[r0,#0]
   0x000080f8   str      r1,[r0,#8]
32
33             //  control register 3 (transmitter and receiver enable)
34             pUART->UTCR3 = 0x00000003;
   0x000080fc   mov      r0,#3
   0x00008100   ldr      r1,0x00008110 ; = #pUART
   0x00008104   ldr      r1,[r1,#0]
   0x00008108   str      r0,[r1,#0xc]
35          }
   0x0000810c   mov      pc,r14
   0x00008110   dcd      0x0000aab8  ....
```

Listing 19. Code for pointer to struct without optimisations

The number of assembly instructions is identical, but now the compiler detects that the destination addresses are just small offsets to one base address (given by the pointer) and encodes the offset in the store instruction. Although the value of r0 is not changed after the first assignment it is still reloaded for every access which costs two unnecessary assembly instructions per register access. Again the last assignment uses swapped register allocation.

Finally the #define version:

```
24          // set up UART1 for 9600 baud with everything enabled to go
25          void define_init(void)
26          {
27             // control register 0 (basic setup)
```

Listing 20. Code for #define without optimisations

```
28              REG(UART1_BASE, UTCR0) = 0x00000008;
   0x00008114   mov      r1,#8
   0x00008118   mov      r0,#0x10000
   0x0000811c   add      r0,r0,#0x80000000
   0x00008120   str      r1,[r0,#0]
29
30              // control register 1 & 2 (baud rate)
31              REG(UART1_BASE, UTCR1) = 0x00000000;
   0x00008124   mov      r1,#0
   0x00008128   nop
   0x0000812c   str      r1,[r0,#4]
32              REG(UART1_BASE, UTCR2) = 0x00000001;
   0x00008130   mov      r1,#1
   0x00008134   nop
   0x00008138   str      r1,[r0,#8]
33
34              // control register 3 (transmitter and receiver enable)
35              REG(UART1_BASE, UTCR3) = 0x00000003;
   0x0000813c   mov      r1,#3
   0x00008140   nop
   0x00008144   str      r1,[r0,#0xc]
36          }
   0x00008148   mov      pc,r14
```

Listing 20. Code for #define without optimisations

Perhaps due to the fact that the resulting address now is passed to the compiler as an expression the compiler recognizes that in each of the four cases the base address is the same. It generates the required four instructions for the first assignment, for the following three assignments it just loads the new value into register r1 and stores that value into the location given by the base address in register r0 and an offset passed as an immediate value.

In this variant the destination address is generated by a MOV-instruction followed by an ADD-instruction due to the limitations of immediate operands described in the section about the load/store instructions (See 6.2.1 “ARM load/store Instructions“). The same result could have been produced by storing the base address as a 32-bit value in the code after the function and loading this value with a LDR-instruction. The total amount of memory used remains the same (two instructions or one instruction and the address value). Using the MOV/ADD-combination it always takes exactly two instruction cycles. Using the LDR-instruction results in only one instruction to execute. The time for this depends on whether the value is already contained in the data cache in which case there is no delay - while the instruction is decoded and prepared for execution the value is fetched from the cache, thus saving one instruction cycle compared to the MOV/ADD-combination. Otherwise the value has to be loaded from memory which can cause considerable delay.

Why the compiler generates the three nop-instructions is not clear - theoretically there is no reason for them as no pipeline stall can happen in this code (and as will be shown later in the optimized version the nops are not generated, they are therefore not required).

6.2.3 No Register Allocation Optimization

Now the optimization level has been changed to “no register allocation optimization”.

Again the code produced for the first to variants is equal:

```
21          // set up UART1 for 9600 baud with everything enabled to go
22          void const_init(void)
23          {
24             // control register 0 (basic setup)
25             *(unsigned int *)const_UT1CR0 = 0x00000008;
   0x00008178   mov      r2,#8
   0x0000817c   ldr      r1,0x000081b4 ; = #const_UT1CR0
   0x00008180   mov      r0,r1
   0x00008184   ldr      r1,[r1,#0]
```

Listing 21. Code for const declaration with “no register allocation”

```
   0x00008188   str      r2,[r1,#0]
26
27              // control register 1 & 2 (baud rate)
28              *(unsigned int *)const_UT1CR1 = 0x00000000;
   0x0000818c   mov      r1,#0
   0x00008190   ldr      r2,[r0,#4]
   0x00008194   str      r1,[r2,#0]
29              *(unsigned int *)const_UT1CR2 = 0x00000001;
   0x00008198   mov      r1,#1
   0x0000819c   ldr      r2,[r0,#8]
   0x000081a0   str      r1,[r2,#0]
30
31              // control register 3 (transmitter and receiver enable)
32              *(unsigned int *)const_UT1CR3 = 0x00000003;
   0x000081a4   mov      r1,#3
   0x000081a8   ldr      r2,[r0,#0xc]
   0x000081ac   str      r1,[r2,#0]
33          }
```

Listing 21. Code for const declaration with “no register allocation”

With this setting the compiler uses a third register (r0) to hold the base address for the variables. This saves one assembly instruction per assignment for all but the first assignment, but adds one instruction in the first assignment. This also implies that an additional register has to be saved to the stack upon entry to the function and restored upon exit. Note the register allocation varies between the first and the three following assignments.

For the struct version again the only difference compared to the `const`/pointer version is that the offset is used in the store-instruction instead of the load-instruction:

```
20          // UART1 port address pointer declaration
21          uart* pUART = (uart*) 0x80010000;  //UART1 Base Address
22
23          // set up UART1 for 9600 baud with everything enabled to go
24          void struct_init(void)
25          {
26             // control register 0 (basic setup)
27             pUART->UTCR0 = 0x00000008;
   0x000080cc   mov      r2,#8
   0x000080d0   ldr      r1,0x00008108 ; = #pUART
   0x000080d4   mov      r0,r1
   0x000080d8   ldr      r1,[r1,#0]
   0x000080dc   str      r2,[r1,#0]
28
29             // control register 1 & 2 (baud rate)
30             pUART->UTCR1 = 0x00000000;
   0x000080e0   mov      r1,#0
   0x000080e4   ldr      r2,[r0,#0]
   0x000080e8   str      r1,[r2,#4]
31             pUART->UTCR2 = 0x00000001;
   0x000080ec   mov      r1,#1
   0x000080f0   ldr      r2,[r0,#0]
   0x000080f4   str      r1,[r2,#8]
32
33             //  control register 3 (transmitter and receiver enable)
34             pUART->UTCR3 = 0x00000003;
   0x000080f8   mov      r1,#3
   0x000080fc   ldr      r2,[r0,#0]
   0x00008100   str      r1,[r2,#0xc]
35          }
   0x00008104   mov      pc,r14
   0x00008108   dcd      0x0000aa94  ....
```

Listing 22. Code for struct declaration with “no register allocation”

In the `#define` version now the nops are removed and regarding the general case the optimal code is being produced - four instructions for the first assignment and only two for each of the following assignments and the code uses only two registers:

```
24                  // set up UART1 for 9600 baud with everything enabled to go
25                  void define_init(void)
26                  {
27                     // control register 0 (basic setup)
28                     REG(UART1_BASE, UTCR0) = 0x00000008;
          0x0000810c   mov      r1,#8
          0x00008110   mov      r0,#0x10000
          0x00008114   add      r0,r0,#0x80000000
          0x00008118   str      r1,[r0,#0]
29
30                     // control register 1 & 2 (baud rate)
31                     REG(UART1_BASE, UTCR1) = 0x00000000;
          0x0000811c   mov      r1,#0
          0x00008120   str      r1,[r0,#4]
32                     REG(UART1_BASE, UTCR2) = 0x00000001;
          0x00008124   mov      r1,#1
          0x00008128   str      r1,[r0,#8]
33
34                     // control register 3 (transmitter and receiver enable)
35                     REG(UART1_BASE, UTCR3) = 0x00000003;
          0x0000812c   mov      r1,#3
0x00008130             str      r1,[r0,#0xc]
36                  }
          0x00008134   mov      pc,r14
```

Listing 23. Code for define declaration with “no register allocation”

In this special case though further improvement is possible. The ARM instruction set allows to read a number of **consecutive** memory locations into a number of registers and to store the contents of multiple registers into **consecutive** memory locations with a single instruction called load/store multiple (`LDM/STM`). Using the `STM` instruction the relevant part of the init function could be encoded as follows:

```
MOV r0, #0x10000
ADD r0, r0, #0x80000000
MOV r1, #8
MOV r2, #0
MOV r3, #1
MOV r4, #3
STMIA r0, {r1, r2, r3, r4}
```

Listing 24. Optimal code for the UART init function

The appended two characters “`IA`” encode the addressing-mode “increment after”, i.e. the first address to write to is taken from the base register (`r0` in this example), the following addresses are formed by incrementing the previous address by four. Optionally the base address register can be updated. The use of this instruction implicates that all the used registers have to be saved and afterwards be restored. As this is done via the stack and the accessed portion of the stack in most cases will be already in the cache the performance penalty should not be too big. Saving/restoring registers on function entry/exit usually is also done using the store/load multiple instructions.

Using the store multiple instruction here is possible, because the accessed four registers happen to be located on consecutive memory locations. In the general case this might not be the case so then `STM` cannot be used.

6.2.4 Full Optimization

For the `const` and the pointer version now the unnecessary fifth instruction in the first assignment is removed. The rest of the code is identical to the version with the option “no memory allocation optimization” set, except that in the last assignment the registers are again swapped:

```
21           // set up UART1 for 9600 baud with everything enabled to go
22           void const_init(void)
23           {
24              // control register 0 (basic setup)
25              *(unsigned int *)const_UT1CR0 = 0x00000008;
   0x00008170   mov      r2,#8
   0x00008174   ldr      r0,0x000081a8 ; = #const_UT1CR0
   0x00008178   ldr      r1,[r0,#0]
   0x0000817c   str      r2,[r1,#0]
26
27              // control register 1 & 2 (baud rate)
28              *(unsigned int *)const_UT1CR1 = 0x00000000;
   0x00008180   mov      r1,#0
   0x00008184   ldr      r2,[r0,#4]
   0x00008188   str      r1,[r2,#0]
29              *(unsigned int *)const_UT1CR2 = 0x00000001;
   0x0000818c   mov      r1,#1
   0x00008190   ldr      r2,[r0,#8]
   0x00008194   str      r1,[r2,#0]
30
31              // control register 3 (transmitter and receiver enable)
32              *(unsigned int *)const_UT1CR3 = 0x00000003;
   0x00008198   mov      r1,#3
   0x0000819c   ldr      r0,[r0,#0xc]
   0x000081a0   str      r1,[r0,#0]
33           }
```

Listing 25. Code for const declaration with full optimisation

For the pointer version this seems to be the optimal code, for the `const` version with analysis of the constant values it should be possible to shrink it down to the same size as the `#define` version.

The struct version again produces the same code as the `const`/pointer version except for putting the offset into the store instruction instead of the load instruction as described before.

For the `#define` version there isn’t any change compared to the “no register allocation optimization” setting as this has already produced the optimal code.

6.2.5 Memory Consumption and Number of Instructions for the Different Declaration Styles

The following table shows the memory map for the three different optimization levels (all numbers in hexadecimal). It shows that the codesize for the `const`, pointer and struct version are the same within each of the optimization levels whereas the codesize for the `#define` version always is considerably smaller. In addition to the codesize the four coding variants use different amount of data space: The pointer and the `const` version use one storage location in the read/write (RAM) and read only (ROM) data segment respectively for each register giving a total of 16 bytes for the given example. The struct version requires one memory location in the read/write data segment while the `#define` version doesn’t use any data space.

	No opt.	no reg.opt.	full opt.		
	Size	Size	Size	Type	Name
const	48	40	3c	CODE RO	C$$code from object file decl_const.o
pointer	48	40	3c	CODE RO	C$$code from object file decl_ptr.o
struct	48	40	3c	CODE RO	C$$code from object file decl_struct.o

Table 22. Memory map

	No opt.	no reg.opt.	full opt.		
define	38	2c	2c	CODE RO	C$$code from object file decl_define.o
const	10	10	10	DATA RO	C$$constdata from object file decl_const.o
pointer	10	4	4	DATA RW	C$$data from object file decl_ptr.o
define	4	4	4	DATA RW	C$$data from object file decl_struct.o

Table 22. Memory map

The following two tables compare the number of assembly instructions necessary for the first access and subsequent accesses for the four different declaration styles:

Style	no opt.	no reg. opt.	full opt.
const	4	5	4
pointer	4	5	4
struct	4	5	4
define	4	4	4

Table 23. Number of instructions for first register access

Style	no opt.	no reg. opt.	full opt
const	4	3	3
pointer	4	3	3
struct	4	3	3
define	3	2	2

Table 24. Number of instructions for subsequent register accesses

6.2.6 Example Illustrating the Necessity of Using the Volatile Keyword

The following polling function is a typical example for the kind of code that requires the variable to be declared as volatile. It polls the UART status register until there is at least one byte available in the receiver FIFO and returns that byte. The first macro definition results in erroneous code unless all optimization is disabled whereas the second version always results in correct code:

```
//#define REG(base,offset) (*(unsigned int*)(base+offset)  // wrong !!
#define REG(base,offset) (*(volatile unsigned int*)(base+offset)  // correct !!

unsigned char read_char(void)
{
   int status;
   do
   {
      status = REG(UART_BASE, UTSR0); // read the UART status register
      status &= 0x2;  // check if bit RNE (receiver not empty)  is set
   }
   while(!status)
   return REG(UART_BASE, UTDR); // read the char from the FIFO and return it
}
```

Listing 26. Read a character from a serial port in polling mode

In the following code fragment the wrong version was used, the pointer was not cast to volatile. The compiler sees an assignment that is independent of the loop variable and therefore places the instructions for loading the register

outside the loop. This results in just loading the (constant) value from r0 to r2 and comparing it with zero. This in effect creates an endless loop if the status bit is not yet set when the function is entered:

```
39          char read_char(void)
40          {
41             int status;
42             do
43             {
   0x00008170          mov      r1,#0x10000
   0x00008174          add      r1,r1,#0x80000000
   0x00008178          ldr      r0,[r1,#0x1c]
   0x0000817c          and      r0,r0,#2
44                status = REG(UART_BASE, UTSR0); // read the UART status register
45                status &= 0x2; // check if bit RNE (receiver not empty) is set
   0x00008180          mov      r2,r0
46             }
47             while(!status);
   0x00008184       cmp      r2,#0
   0x00008188       beq      0x8180  ; (read_char + 0x10)
48             return REG(UART_BASE, UTDR); // read a char from the FIFO and return it
   0x0000818c       ldr      r0,[r1,#0x14]
   0x00008190       and      r0,r0,#0xff
   0x00008194       mov      pc,r14
49          }
```

Listing 27. Erroneous code

Now the correct version was used, the instructions to load the value from the UART status register into register r0 and the and-operation is included in the loop and the code works as expected.

```
51          char read_char(void)
52          {
53             int status;
54             do
55             {
   0x00008198          mov      r1,#0x10000
   0x0000819c          add      r1,r1,#0x80000000
56                status = REG(UART_BASE, UTSR0); // read the UART status register
   0x000081a0          ldr      r0,[r1,#0x1c]
57                status &= 0x2; // check if bit RNE (receiver not empty) is set
   0x000081a4          and      r0,r0,#2
58             }
59             while(!status);
   0x000081a8       cmp      r0,#0
   0x000081ac       beq      0x81a0  ; (read_char + 0x8)
60             return REG(UART_BASE, UTDR); // read a char from the FIFO and return it
   0x000081b0       ldr      r0,[r1,#0x14]
   0x000081b4       and      r0,r0,#0xff
   0x000081b8       mov      pc,r14
61          }
```

Listing 28. Correct code using volatile

6.2.7 Conclusions

Clearly the `const` and the pointer versions should be avoided. Their drawbacks become even more obvious when considering the use of multiple peripherals. For each device used one variable per peripheral register is required. The `struct` version allows the programmer to write straight forward C-source code, provides encapsulation and it allows switching the device by just changing the base pointer. If macros are used with the `#define` version the source code is still easily readable and offers by far the smallest codesize. In the targeted type of applications, involving a lot of I/O-accesses in embedded devices having very limited resources, this can save a considerable amount of valuable memory space and execution time and therefore also power.

7. Debugging Embedded Systems

This chapter describes some of the problems related to program development and debugging which I faced during my work with the SmartBadge. It then presents some techniques I used to overcome and circumvent these problems and the limitations of the environment being used.

7.1 Description of Used Tools

During my work I used the ARM Software Development Toolkit Version 2.11 for Windows. This package contains a project manager which integrates an editor and project management (managing source files within a project, setting compiler options for the project, a target or single files, setting linker options,...), compiler, assembler, linker, and a debugger.

In general the debugger is quite comfortable - it offers most of the features one expects from a modern debugger. Some examples are:

- setting watchpoints and breakpoints (optionally with conditions)
- interleaved display, i.e. showing the C-source code and for each line of source code showing the generated assembly code. In this mode single stepping on assembly instruction level is possible.
- display of the current CPU register values
- display of local and global variables, in the case of pointers also displaying the value at the location the pointer points to. If the variable is a structure type it is also possible to display its member variables.
- display of memory regions
- disassembler window
- expression evaluation, can use preprocessor defines (but not macros)
- a console window to read and write characters from and to the target. This makes use of the semihosted C-library which diverts standard I/O (printf, fprintf, scan, scanf,...) from the target to the host running the debugger.

7.2 Problems Related to the ARM debugger

On the other hand there are some drawbacks which impair working with this debugger. Especially two problems turned out to be really annoying:

- Although it is possible to save a configuration file for the debugger this contains only options for the command line debugger, but not for the graphical user interface. In practice this means that it is possible to save breakpoints, but not for example the entered expressions or the base address for memory windows. So after each debugger restart one has to reenter all the expressions and open the needed memory windows.
- The debugger needs to maintain continuous communication with the target. If the target crashes or is reset the debugger crashes and has to be killed with the Windows NT task manager.

These two problems together turned out to be very time wasting especially when working with the StrongARM peripherals in combination with interrupts. Troubleshooting the peripherals typically requires watching a number of peripheral registers. Using the simple method - displaying the contents in a memory window poses problems when FIFOs are involved. If for example the memory window is set to offset 0x80010000 - the base address of UART1 - all the registers of UART1 can conveniently be observed. But the UART data register is located between the control registers and the status registers. On every update of the memory window all the displayed memory locations are read which in that case means also reading the UART data register. Reading the UART data register removes the top entry from the FIFO which then is not available for the actual program running on the target any more. Therefore the program misses much of the data it is supposed to process.

One solution is to use the expression evaluation window and enter only the required control and status registers. These expressions are kept over multiple debugging sessions (stopping a program and reloading / executing it), but not when the debugger is closed (or what happens most of the time - has to be killed). So whenever the target being debugged crashes (which tends to happen quite often during the development process, especially if multiple interrupts and peripherals are used) or has to be reset to get into a defined starting state the debugger also crashes and has to be killed. After restart all the expressions have to be reentered (including e.g. setting the display format to hex for every single expression). Another problem is that the results of the evaluated expressions are not always updated properly in single step mode - most of the time they are updated whenever their values change but sometimes this update only occurs with an explicit selection/double click on the expression.

7.3 General Problems in Debugging Embedded Systems

Apart from the problems described in the previous section - which are clearly due to a lack of functionality or are even errors in the implementation of the debugger - there are some general limitations which are inherent to the problem of debugging embedded systems:

- typically embedded systems have only limited means of communication (passing on the desired information to the external world). During my work I made heavy use of the second serial port to pass information which could not be acquired by using the debugger. On the PC side there could be a simple terminal program like Kermit or Hyperterminal to display status messages or a specialized program -- if for example link frames, hexadecimal data, timing information, or something similar has to be evaluated or verified.
- there are general problems in debugging interrupts: In many systems, among them the StrongARM, at least when using the Angel Debug Monitor (the part of the debugger which resides on the target) all other interrupts are disabled while one interrupt is being processed. As the debugger itself uses an interrupt (for UART3) to communicate with the target, it is not possible to execute an interrupt routine under control of the debugger without crashing the session.

  Unlike my expectations this **also** applies to code executed via the Angel queuing mechanism (See “Extensions to Angel” on page 16), although this code is executed in user mode with interrupts enabled. This effectively made debugging (using the debugger functions) impossible for nearly all of the code in the IrDA project (Chapter 8. on page 131) as most of that code is executed via the queuing mechanism as result of timer or communication events.
- inherent problems when time dependant processes are to be analysed: clearly the delay introduced by single stepping through code is not tolerable if certain actions in the target program have to occur within specified time constraints.

7.4 Solutions

Generally one can distinguish between three methods to debug an embedded system:

- Use a debugger if no interrupts or timing sensitive processes are involved.
- Use additional debugging code to keep track of the instruction flow or variable values, mostly output via an additional, otherwise unused (serial) port.
- Use an oscilloscope or a logic analyzer to observe single signals or even the whole address/data bus on the hardware level. This provides the most accurate information, but also is the most expensive solution. Depending on the features offered by the logic analyzer the results can vary from a series of hexadecimal values representing the values of the observed buses to assembly code output automatically generated based on the acquired binary values if processor specific logic analyzer modules are used. In any case the use of a logic analyzer provides the most exact information about the processor activity - especially for debugging low level code, for instance in operating systems, this is often the only solution. However, if caches are enabled, not even a logic analyzer can reveal all details, as much of the activity then only occurs within the chip and is not visible from outside.

In the following I will show a few examples using method two - additional debug code - which I used during my work with the SmartBadge.

7.4.1 Counting Variables

To verify the operation of the SIR modulation interrupt routine I used counters for each of the sections. By transmitting a known character sequence, I could precompute the final expected values for each of the counters and compare these with the actual values after program execution. Although this method does not reveal any information about the sequence in which the sections were executed, it tells at least how often each section was passed and this was often enough information to detect the initial errors.

Note: For this application the timing also was extremely important. To verify the correct timing of the software modulation of the IR emitter I used a logic analyzer to measure and verify the output signal of the PPC pin (TXD2) which is connected to the off-chip infrared transceiver module.

7.4.2 Writing Debug Data to a Buffer

If more information than simple counters can provide is needed, e.g. information about the sequence in which particular parts of the code are executed, one can use a debug buffer. During program execution at the interesting locations data is written to that buffer. At a breakpoint or before the end of the program this buffer can be output and thus reveal the detailed program flow.

Both methods have a severe drawback: they only work if the program does not crash, i.e. they allow verification of the program flow and thus often enable the user to detect errors. But they can’t reveal any information about the actual problem if the program crashes before the output point is reached.

The advantage of both these methods is that their overhead - especially in terms of execution time (no waiting on devices) - is very small in comparison to the buffered output described in the next section.

7.4.3 Debug Output to a Serial Port

As serial port three is taken by the Debugger/Angel Debug Monitor I used serial port one to output additional information. This method allows three slightly different approaches - buffered output, polled output and direct output:

For direct output the desired output byte is written directly to the UART data register. The advantage of this method is that there is minimal overhead and the output occurs nearly immediately. The problem is of course that debug output can be lost if the write operations occur faster than the serial port can transmit.

The code for this type of output is simply a peripheral register write (after the UART has been initialized):

```
#define UART1_BASE 0x80010000
#define UTDR 0x14

#define DEBUG_UART UART1_BASE

REG(DEBUG_UART, UTDR) = ch;     // output character ‘ch’ on UART 1
```

Listing 29. Direct debug output via a serial port

For buffered output I used a ring buffer structure and interrupt driven output. To generate debug output the desired characters are written to the ring buffer and when this write operation is finished the serial transmit interrupt is enabled. This method guarantees that no output is lost, but the output can be delayed - if there are other, higher priority interrupts pending, the serial interrupt remains pending until all higher prioritized interrupts have been executed. In the worst case the output might never happen if the frequency of higher priority interrupts is too high (or the program crashes/does not return from an interrupt/supervisor mode with interrupts disabled).

Listings 30 and 31 show the declarations and the code related to dealing with the ring buffer structures:

```
/*
 *      Description:
 *          Declarations to manipulate ring buffers.
 *          The source is derived from code in ringbuff.h, but was
 *          modified to allow user defined buffer size and atomic access for
 *          concurrent use in interrupt handler and regular program code.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */


#ifndef util_ringbuf_h
#define util_ringbuf_h

#include <stdlib.h>     /* for malloc */
#include “util/util_misc.h”


/*
 * the ring buffer structure
 */
typedef struct RingBuffer
{
    volatile unsigned int size;  // the size of the allocated memory region
    volatile unsigned int head;  // offset to current beginning of the buffer
    volatile unsigned int tail;  // offset to current end of the buffer
    volatile UC8* data;          // pointer to the allocated memory region
} RingBuffer;


/*
 * ring buffer operations as macros
 * Note that these will overflow after MAXUINT characters
 */

// only to be used in interrupts as it assumes exclusive access  !
#define ringbuf_GetCountInt(r)  ( ABS( ((r)->head - (r)->tail) % (r)->size ) )


#define ringbuf_NotEmpty(r)     ( ringbuf_GetCount(r) > 0 )
#define ringbuf_Empty(r)        ( ringbuf_GetCount(r) == 0 )
#define ringbuf_Full(r)         ( ringbuf_GetCount(r) >= (r)->size )
#define ringbuf_GetSpace(r)     ( (r)->size - ringbuf_GetCount(r) )
#define ringbuf_ReadByte(r)     ( (r)->data[((r)->tail++) % (r)->size] )
#define ringbuf_WriteByte(r, c) ( (r)->data[((r)->head++) % (r)->size] = (c) )


// free the memory allocated to a ring buffer structure
#define ringbuf_Free(r) (free((r)->data))
```

Listing 30. Ring buffer declarations [file util\util_ringbuf.h]

```
/*
 *      Description:
 *          Functions to manipulate ring buffers.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */
```

Listing 31. Ring buffer functions [file util_ringbuf.c]

```
#include “util/util_ringbuf.h”
#include “util/util_interrupt.h”
#include “util/util_misc.h”


/* to check if the Angel function hooks have been initialized */
extern int misc_initialized;


/*
 * Function: ringbuf_Init
 *  Purpose: Initialize a ringbuffer
 *
 * Parameters:
 *      Input: buf              pointer to the buffer to be initialized
 *             size             desired size of the buffer
 *
 *    Returns: 0                success
 *             -1               memory allocation failed
 *
 * The function tries to allocate ‘size’ bytes of space for the buffer ‘buf’
 * and initializes the fields.
 */
int ringbuf_Init(RingBuffer* buf, UI32 size)
{
    // hook to Angel_EnterSVC() is needed in one of the functions, therefore
    // initialize hook functions if that hasn’t been done yet.
    if(!misc_initialized)
    {
        misc_InitAngelFunctions();
    }

    // allocate the memory for the buffer and initialize the other fields
    buf->data = (UC8*)malloc(size);
    if(buf->data)
    {
        buf->size = size;
        buf->head= 0;
        buf->tail=0;
        return 0;
    }
    else
        return -1;
}


/*
 * Function: ringbuf_GetCount
 *  Purpose: get the number of bytes in a ring buffer
 *
 * Parameters:
 *      Input: buf              pointer to the buffer
 *
 *    Returns: the number of bytes in the buffer
 *
 * Get number of bytes that are currently in the buffer.
 * As opposed to the macro ‘ringbuf_GetCountInt(r)’ which is intended
 * for use in interrupt handlers only, this function disables all interrupts
 * before computing the result ‘count’ in order to guarantee atomic access
 * to buf->head and buf->tail which could be modified by an interrupt
 * otherwise.
 */
int ringbuf_GetCount(RingBuffer* buf)
```

Listing 31. Ring buffer functions [file util_ringbuf.c]

```
{
    int count;

    Angel_EnterSVC();   // disable interrupts
    count = ABS( (buf->head - buf->tail) % buf->size );
    Angel_ExitToUSR();  // enable interrupts
    return count;
}


/*
 * Function: ringbuf_WriteBuf
 *  Purpose: write the contents of a byte array to a ring buffer
 *
 * Parameters:
 *      Input: buf              pointer to the ring buffer to write to
 *             data             pointer to the source array
 *             len              number of bytes to be written
 *
 *    Returns: void
 *
 * Write 'len' bytes of data to the buffer 'buf'. The caller must
 * check in advance that there is enough space in the buffer, otherwise
 * it will wrap around and overwrite contents at the "beginning" of the
 * buffer.
 */
void ringbuf_WriteBuf(RingBuffer *buf, const UC8* data, UI32 len)
{
    int i;
    for(i=0;i<len;i++)
    {
        ringbuf_WriteByte(buf, *(data+i));
    }
}


/*
 * Function: ringbuf_WriteBufCheck
 *  Purpose: write the contents of a byte array to a ring buffer, check for
 *           sufficient space.
 *
 * Parameters:
 *      Input: buf              pointer to the ring buffer to write to
 *             data             pointer to the source array
 *             len              number of bytes to be written
 *
 *    Returns: the number of bytes actually written
 *
 * Write max(len, available bufferspace) bytes of data to the specified
 * buffer buf. The number of bytes actually written is returned.
 */
int ringbuf_WriteBufCheck(RingBuffer *buf, const UC8* data, UI32 len)
{
    int i;
    len = MIN(len, ringbuf_GetSpace(buf));
    for(i=0;i<len;i++)
    {
        ringbuf_WriteByte(buf, *(data+i));
    }
    return len;
}


/*
 * Function: ringbuf_ReadBuf
```

Listing 31. Ring buffer functions [file util_ringbuf.c]

```
 *  Purpose: read bytes from a ring buffer into a byte array
 *
 * Parameters:
 *      Input: buf              pointer to the ring buffer to read from
 *             data             pointer to the destination array
 *             len              number of bytes to be transferred
 *
 *    Returns: void
 *
 * Read len bytes of data out of the specified buffer buf and write them to
 * 'data'. The caller must check in advance that the buffer contains at least
 * 'len' bytes and must allocate the memory for data.
 */
void ringbuf_ReadBuf(RingBuffer *buf, UC8* data, UI32 len)
{
    int i;
    for(i=0;i<len;i++)
    {
        *(data+i) = ringbuf_ReadByte(buf);
    }
}


/*
 * Function: ringbuf_ReadBufCheck
 *  Purpose: read bytes from a ring buffer into a byte array, check for
 *           number of bytes
 *
 * Parameters:
 *      Input: buf              pointer to the ring buffer to read from
 *             data             pointer to the destination array
 *             len              number of bytes to be transferred
 *
 *    Returns: the number of actually read bytes
 *
 * Read max(len, available bytes in 'buf') bytes out of 'buf' and write them
 * to 'data'. The caller must allocate the memory for data.
 * The number of actually read bytes is returned.
 */
int ringbuf_ReadBufCheck(RingBuffer *buf, UC8* data, UI32 len)
{
    int i;
    len = MIN(len, ringbuf_GetCount(buf));
    for(i=0;i<len;i++)
    {
        *(data+i) = ringbuf_ReadByte(buf);
    }
    return len;
}
```

Listing 31. Ring buffer functions [file util_ringbuf.c]

The code I used for debugging is as follows, listing 32 shows the declarations and inline functions, listing 33 shows the debug functions:

```
/*
 *      Description:
 *          Type declarations for debug functions and inline functions.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */
```

Listing 32. Debug declarations and inline functions [util\util_debug.h]

```
#ifndef util_debug_h
#define util_debug_h

#include <util/util_misc.h>
#include <util/util_ringbuf.h>
#include <util/util_serial.h>

#include <stdio.h>


// set the default debug level - DEBUG_LEVEL can be defined per source file
// (must be set before including this header file)
#ifndef DEBUG_LEVEL
    #define DEBUG_LEVEL 4
#endif


// the debug buffer
extern RingBuffer debug_tx_buf;


// the UART to be used for the debug output
#define DEBUG_UART_BASE UART1_BASE
#define DEBUG_UART_INT  INT_UART1


/*
 * Function: debug_Init
 *  Purpose: Setup the debug UART channel.
 *
 * Parameters: none
 *    Returns: void
 *
 * The function installs the debug interrupt handler, initializes the debug
 * transmission buffer and sets up the UART channel for 115KBaud, 8N1.
 */
void debug_Init(void);


/*
 * Function: debug_String
 *  Purpose: Output a string on the debug UART channel.
 *
 * Parameters:
 *      Input: string   The string to be output
 *
 *    Returns: void
 *
 * The function outputs the given null-terminated string on the UART channel
 * specified by DEBUG_UART_BASE by writing it to the debug buffer. If the
 * buffer does not contain enough space its contents will be overwritten.
 * After the bytes have been written the transmit interrupt is enabled to
 * start the transmission.
 */
void debug_String(char* string);

void debug_BufferHex(UC8* buf, int len);

/*
 * Function: debug_PutByteDirect
 *  Purpose: Write a byte directly to the debug FIFO
 *
 * Parameters: c        the byte to output
 *    Returns: void
 *
```

Listing 32. Debug declarations and inline functions [util\util_debug.h]

```
 * The macro writes directly to the FIFO, thus output can be lost if there
 * is not enough space in the FIFO.
 */
#define debug_PutByteDirect(c) REG(DEBUG_UART_BASE, UTDR) = (c)


/*
 * Function: debug_PutByte
 *  Purpose: Write a byte to the debug buffer.
 *
 * Parameters: ch       the byte to output
 *    Returns: void
 *
 * Write a single byte to the debug buffer, does not start the transmission.
 */
__inline void debug_PutByte(UC8 ch)
{
    ringbuf_WriteByte(&debug_tx_buf, ch);
}


/*
 * Function: debug_PutByteHex
 *  Purpose: Write a byte in hexadecimal format to the debug buffer.
 *
 * Parameters: ch       the byte to output
 *    Returns: void
 *
 * Write a single byte in hexadecimal format to the debug buffer, does not
 * start the transmission.
 */
__inline void debug_PutByteHex(UC8 ch)
{
    char buf[4];

    sprintf(buf, "%2.2x ", ch);
    ringbuf_WriteByte(&debug_tx_buf, buf[0]);
    ringbuf_WriteByte(&debug_tx_buf, buf[1]);
    ringbuf_WriteByte(&debug_tx_buf, buf[2]);
}


/*
 * Function: debug_PutBytePolled
 *  Purpose: Write a byte to the FIFO in polling mode
 *
 * Parameters: ch       the byte to output
 *    Returns: void
 *
 * Write a single byte to the FIFO, but poll the TNF flag until there
 * is at least one free position in the FIFO to make sure no output is
 * lost.
 */
#define debug_PutBytePolled(c) ser_UartPutBytePolled(DEBUG_UART_BASE, c)



/*
 * Function: debug_PutByteHex
 *  Purpose: Write a byte in hexadecimal format to the FIFO in polling mode
 *
 * Parameters: ch       the byte to output
 *    Returns: void
 *
 * Write a single byte in hexadecimal format to the FIFO, but poll the TNF
```

Listing 32. Debug declarations and inline functions [util\util_debug.h]

```
 * flag until there is at least one free position in the FIFO to make sure
 * no output is lost.
 */
__inline void debug_PutByteHexPolled(UC8 ch)
{
    char buf[5];

    sprintf(buf, "0x%2.2x ", ch);
    debug_PutBytePolled(buf[0]);
    debug_PutBytePolled(buf[1]);
    debug_PutBytePolled(buf[2]);
    debug_PutBytePolled(buf[3]);
    debug_PutBytePolled(buf[4]);
}


/* Debug macros that are only compiled and executed if DEBUG_LEVEL is higher
 * or equal than their first argument
 */

// output string s if DEBUG_LEVEL is higher or equal than n
#define DEBUG(n,s)  debstr ## n(s "\r")

// output string s with argument v if DEBUG_LEVEL is higher or equal than n
// (like printf with one variable argument supported)
#define DEBUG_1(n,s,v) debstr_m ##n(s "\r",v)
#define debstr_m(s,v)   { char buf[100]; sprintf(buf, s, v); debug_String(buf);}


// code to achieve the conditional compilation depending on an argument
#if (DEBUG_LEVEL == 0)
    #define debstr0(str)    debug_String(str)
    #define debstr1(str)
    #define debstr2(str)
    #define debstr3(str)
    #define debstr4(str)

    #define debstr_m0(s,v)  debstr_m(s,v)
    #define debstr_m1(s,v)
    #define debstr_m2(s,v)
    #define debstr_m3(s,v)
    #define debstr_m4(s,v)

#elif (DEBUG_LEVEL == 1)
    #define debstr0(str)    debug_String(str)
    #define debstr1(str)    debug_String(str)
    #define debstr2(str)
    #define debstr3(str)
    #define debstr4(str)

    #define debstr_m0(s,v)  debstr_m(s,v)
    #define debstr_m1(s,v)  debstr_m(s,v)
    #define debstr_m2(s,v)
    #define debstr_m3(s,v)
    #define debstr_m4(s,v)

#elif (DEBUG_LEVEL == 2)
    #define debstr0(str)    debug_String(str)
    #define debstr1(str)    debug_String(str)
    #define debstr2(str)    debug_String(str)
    #define debstr3(str)
    #define debstr4(str)

    #define debstr_m0(s,v)  debstr_m(s,v)
    #define debstr_m1(s,v)  debstr_m(s,v)
```

Listing 32. Debug declarations and inline functions [util\util_debug.h]

```
    #define debstr_m2(s,v)  debstr_m(s,v)
    #define debstr_m3(s,v)
    #define debstr_m4(s,v)

#elif (DEBUG_LEVEL == 3)
    #define debstr0(str)    debug_String(str)
    #define debstr1(str)    debug_String(str)
    #define debstr2(str)    debug_String(str)
    #define debstr3(str)    debug_String(str)
    #define debstr4(str)

    #define debstr_m0(s,v)  debstr_m(s,v)
    #define debstr_m1(s,v)  debstr_m(s,v)
    #define debstr_m2(s,v)  debstr_m(s,v)
    #define debstr_m3(s,v)  debstr_m(s,v)
    #define debstr_m4(s,v)

#else
    #define debstr0(str)    debug_String(str)
    #define debstr1(str)    debug_String(str)
    #define debstr2(str)    debug_String(str)
    #define debstr3(str)    debug_String(str)
    #define debstr4(str)    debug_String(str)

    #define debstr_m0(s,v)  debstr_m(s,v)
    #define debstr_m1(s,v)  debstr_m(s,v)
    #define debstr_m2(s,v)  debstr_m(s,v)
    #define debstr_m3(s,v)  debstr_m(s,v)
    #define debstr_m4(s,v)  debstr_m(s,v)
#endif

#define ERROR(s) debug_String("ERROR: " s "\r")
#define WARNING(s) debug_String("WARNING: " s "\r")

#endif
```

Listing 32. Debug declarations and inline functions [util\util_debug.h]

```
/*
 *      Description:
 *          Code to generate debug output on a UART channel on the Badge.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#include <util/util_debug.h>
#include <util/util_ringbuf.h>
#include <util/util_serial.h>
#include <util/util_interrupt.h>

#include <stdio.h>
#include <string.h>

#define DEBUG_TX_BUFSIZE 4096
#define DEBUG_RX_BUFSIZE   30
```

Listing 33. Debug functions [util_debug.c]

```
/* Global debug buffer to write debug output to */
RingBuffer debug_tx_buf;

/* for other modules to check if debug code has been set up */
int debug_initialized = FALSE;

/*
 * Function: debug_UartIntHandler
 *  Purpose: Debug interrupt handler to output characters in the debug buffer.
 *
 * Parameters: Interrupt handler parameters filled in by Angel
 *    Returns: void
 *
 * The handler only handles the break conditions and the transmitter FIFO
 * request. All other interrupt reasons cause the interrupt to be masked.
 */
void debug_UartIntHandler(unsigned int ident, unsigned int data, unsigned int
empty_stack)
{
    int i;
    int count;

    if(TEST_BIT(DEBUG_UART_BASE, UTSR0, UTSR0_TFS))    // transmit FIFO request
    {
        // get number of characters left in the debug buffer
        count = ringbuf_GetCountInt(&debug_tx_buf);

        // at least four bytes are ready and can be written without further
        // checking
        if(count>=4)
        {
            REG(DEBUG_UART_BASE, UTDR) = ringbuf_ReadByte(&debug_tx_buf);
            REG(DEBUG_UART_BASE, UTDR) = ringbuf_ReadByte(&debug_tx_buf);
            REG(DEBUG_UART_BASE, UTDR) = ringbuf_ReadByte(&debug_tx_buf);
            REG(DEBUG_UART_BASE, UTDR) = ringbuf_ReadByte(&debug_tx_buf);
        }
        else
        {
            for(i=0;i<count;i++)
                REG(DEBUG_UART_BASE, UTDR) = ringbuf_ReadByte(&debug_tx_buf);

            // debug output finished, disable the interrupt
            SER_UART_DISABLE_TI(DEBUG_UART_BASE);
        }
    }
    else if(TEST_BIT(DEBUG_UART_BASE, UTSR0, UTSR0_RBB)) // receiver break begin
    {
        REG(DEBUG_UART_BASE, UTSR0) &= ~UTSR0_RBB;
    }
    else if(TEST_BIT(DEBUG_UART_BASE, UTSR0, UTSR0_REB))    // receiver break end
    {
        REG(DEBUG_UART_BASE, UTSR0) &= ~UTSR0_REB;
    }
    else
    {
        INT_MASK(DEBUG_UART_INT);
    }

}


/*
 * Function: debug_Init
 *  Purpose: Setup the debug UART channel.
```

Listing 33. Debug functions [util_debug.c]

```
 *
 * Parameters: none
 *    Returns: void
 *
 * The function installs the debug interrupt handler, initializes the debug
 * transmission buffer and sets up the UART channel for 115KBaud, 8N1.
 */
void debug_Init(void)
{

    int_InstallHandler(DEBUG_UART_INT, debug_UartIntHandler);

    if(ringbuf_Init(&debug_tx_buf, DEBUG_TX_BUFSIZE) == -1)
    {
        printf("debug error: no memory for debug_tx buffer available\n");
        exit(1);
    }

    ser_UartInit(DEBUG_UART_BASE, SER_INIT_BAUD_115200, SER_INIT_PARITY_DIS,
                  SER_INIT_PARITY_ODD, SER_INIT_ONE_STOP_BIT,
                  SER_INIT_DATA_SIZE_8, SER_INIT_RECEIVE_INT_DIS,
                  SER_INIT_TRANSMIT_INT_DIS);

    // enable transmitter and unmask the first level interrupt, the transmit
    // interrupt enable bit is set when transmission is about to start
    SER_UART_ENABLE_T(DEBUG_UART_BASE);
    INT_UNMASK(DEBUG_UART_INT);

    debug_initialized = TRUE;
}


/*
 * Function: debug_String
 *  Purpose: Output a string on the debug UART channel (interrupt driven).
 *
 * Parameters:
 *      Input: string   The string to be output
 *
 *    Returns: void
 *
 * The function outputs the given null-terminated string on the UART channel
 * specified by DEBUG_UART_BASE by writing it to the debug buffer. If the
 * buffer does not contain enough space its contents will be overwritten.
 * After the bytes have been written the transmit interrupt is enabled to
 * start the transmission.
 */
void debug_String(char* string)
{
    int i;
    int len = strlen((char*)string);
    for(i=0;i<len;i++)
    {
        ringbuf_WriteByte(&debug_tx_buf, string[i]);
    }
    SER_UART_ENABLE_TI(DEBUG_UART_BASE);
}
```

Listing 33. Debug functions [util_debug.c]

Especially when debugging the IrDA stack this method worked great to follow and check the program execution. But it did not help in the cases where the program crashed at some point: due to the limited baud rate the debug output always was far behind the actual execution point and, as a crash also stopped the debug output, I could not tell where exactly in the code the crash had happened. As a consequence of the big amount of debug data (the debug output for

only establishing the IrDA-link was usually more than 600 lines) direct output could not be used because nearly all of the directly output bytes were lost.

That's where the polled output came into play: it still outputs only single bytes like direct output but instead of directly writing to the serial FIFO it polls the TNF (transmitter not full) flag which when set signals that there is at least one empty position in the FIFO. As the transmit request interrupt is only enabled if there are at least four empty positions in the FIFO this method always succeeds. As the output occurs almost immediately (at most it takes the time to transmit one byte) and no other code in the current function will be executed before the output occurs, the fact that a particular debug output byte is not received tells definitely that the program crashed before reaching this point. If the crash happens within a function/interrupt routine under user control one can guard the section causing the problem with polled debug statements and by incrementally moving the statements one can finally isolate the line which is responsible for the crash. As this output is interleaved with the ongoing buffered output one should use special characters which are not used in the standard debug output and therefore can be easily searched for and detected. Compared to the direct output, polled output is preferable in most situations, only in highly time-critical code (where even polling for one FIFO entry to become free could disturb) direct output has to be used.

The following defines are used in the files to be debugged:

```
// in files to be debugged
#define file_name_DEBUG
#ifdef file_name_DEBUG
    #define DEBUG_STRING(c) debug_String(c)
    #define DEBUG_PUT_BYTE(c) debug_PutByte(c)
    #define DEBUG_PUT_BYTE_HEX(c) debug_PutByteHex(c)
#else
    #define DEBUG_STRING(c)
    #define DEBUG_PUT_BYTE(c)
    #define DEBUG_PUT_BYTE_HEX(c)
#endif
```

Listing 34. Defines to be put in files to support debug output

This allows to turn on/off debug output on a per-file base.

A few comments on the code:

- The function `ringbuf_GetCount()` uses the function pair `Angel_EnterSVC()` and `Angel_ExitToUSR()`to disable/enable interrupts and therefore requires that `misc_InitAngelFunctions()` (See 4.1 "Extensions to Angel") has been executed before. If this has not been done explicitly, this function is called from `ringbuf_Init()`.
- `int_InstallHandler()` and `ser_UartInit()` are the functions described in Section 4.8.
- I used an additional `#define` part in every source file I intended to debug (See listing 34) to allow simple inclusion/exclusion of the debug code on a per-file basis.
- If interrupt driven debug output is possible - the user must remember that the output does not give absolute timing information, but only information about the sequence of code execution. Still this was particularly useful in the process of implementing and verifying the software timer functionality (Section 4.7, "Operating System Timers," on page 56), especially to verify that the Angel function calls related to serialization operated as expected. `DEBUG_PUT_BYTE()` statements in each of the functions allowed a detailed trace.

7.5 Conclusions

As long as no interrupts are involved debugging is fairly easy and allows (relatively) fast identification of errors. As soon as interrupts come into play things become *much* more complicated. If the program terminates properly the debugging methods described above are pretty usable and generally allow the user to get enough information to identify any remaining problems.

If a program does not terminate or is time sensitive, one can only rely on the direct/polled debug output which can give some hints, but frequently still leaves a lot of questions open, due to the possible loss of characters (in the case of direct output) and the very limited amount of information (i.e. single bytes) that can be output with this method. In some cases, especially related to timing problems/race conditions, even using polled output might not reveal the problem if it’s not possible to determine the function or the circumstances that cause the crash.

8. IrDA Protocol Stack

Due to its high computing power the SmartBadge can do a lot of operations locally. The range of possible applications is greatly increased if (wireless) network access can be obtained. Basically the Badge offers two ways to achieve this, either using a PCMCIA form factor wireless LAN card or using the infrared port. After a short introduction which compares the two methods I will explain the IrDA protocol stack. A port of the Linux IrDA stack to the SmartBadge, which is the main result of my work, is presented in chapter 9..

8.1 Comparison Wireless LAN - Infrared link

As mentioned in the introduction to this chapter, the SmartBadge offers two major alternatives for (wireless) network connectivity:

- by inserting a wireless LAN (such as the Lucent Technologies IEEE 802.11b Wireless LAN) card into the PCMCIA socket and/or
- by using the infrared transceiver connected to serial port 2

The biggest advantage of an IEEE 802.11b wireless LAN (WLAN) over infrared is the fact that it works without direct line of sight between the Badge and the WLAN base station, thus complete freedom of movement of the Badge is possible. Depending on the environmental conditions (wall and floor materials, electromagnetic noise,...) the range is specified to be up to 25m in a closed office environment, 50m in a semi-open environment, and up to 160m in an open office environment at 11Mb/s.

On the other hand, depending on the type of application, the fact that infrared communication needs direct line of sight between the communicating devices, can also be an advantage - as this provides implicit location information. With a WLAN one can detect that a device is within range of the base station (or another device when ad-hoc mode is used), but no direction dependant information is available. In addition, the distance can not easily be determined (at least not with standard device drivers). Even if the signal strength can be observed, it is hard to determine if a weak signal results from a large distance to the base station or just a thick wall between the device and a nearby basestation. In the case of (non-diffuse) infrared on the other hand, if there is link connectivity, then one knows that both devices are in the same room and are orientated towards each other. The rough distance can be estimated as there must be direct line of sight (which provides geometric limits) and if the signal strength is known, then the location can be even further refined.

The current generation of WLAN cards supports speeds of up to 11Mb/s. Nominally this is considerably greater than the fastest current infrared mode (4Mb/s, but a 16Mb/s mode for infrared is currently being developed), but one has to take into account that all users in the same channel have to share this bandwidth. Whereas in the infrared case the 4 Mb/s are exclusively given to one user/device (or at least only the devices which can fit into the viewable solid angle of the IR receiver).

An additional drawback of the WLAN technology is its relatively high power consumption, although new 3.3V technology can substantially reduce the power consumption. Even so the power needs for a PCMCIA card plus the radio link are higher than those for the infrared transceiver. Additionally of course the amount of space needed for a PCMCIA card is much higher than for a single infrared-transceiver.

Of course it is possible to combine the two technologies: one can use (many small, cheap) infrared beacons to provide location information and, if no infrared access point is in sight and communication is required, then use a WLAN card for network access.

8.2 Overview of the IrDA Protocol Stack

The IrDA protocol stack is a set of protocols, defined by the Infrared Data Organization (http://www.irda.org). Figure 23 gives an overview of the protocol stack:

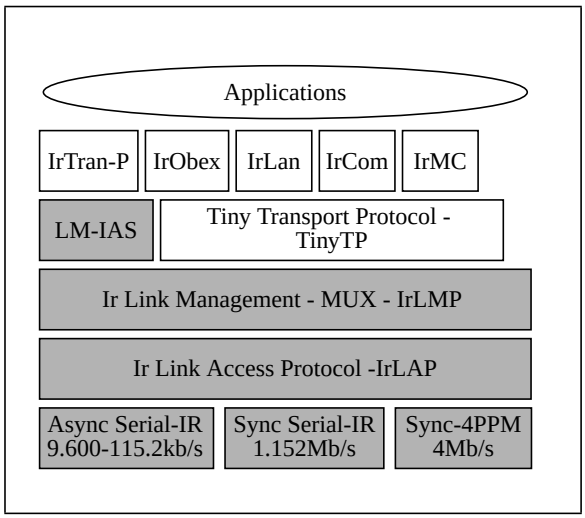


Figure 23. IrDA protocol stack [16]

The stack consists of a set of mandatory protocols (shown in gray in Figure 23) and additional optional protocols. The mandatory protocols are the following:

- IrPHY (Physical Signaling Layer), specified in [11]
- IrLAP (Link Access Protocol), specified in [12]
- IrLMP (Link Management Protocol and Information Access Service (IAS)), specified in [13]

IrPHY is equivalent to OSI layer 1 and defines the hardware specification, including signal modulation. On top of IrPHY IrLAP defines the link layer. It is based on the HDLC- and SDLC-half duplex protocols by IBM and is equivalent to OSI layer 2. IrLAP provides a reliable link between two devices. Corresponding to OSI layer 3, IrLMP is defined on top of IrLAP. It consists of the Link Management Protocol and the Information Access Service. The Link Management Protocol is responsible for multiplexing simultaneous connections via the single IrLAP link. The Information Access Service provides information about registered services which can be queried by remote devices.

To provide LAN access via the IrDA stack, two additional optional protocols are required - the Tiny Transport Protocol (provides flow control on IrLMP connections with an optional Segmentation and Reassembly service) and the IrLan Protocol (encapsulates ethernet frames to be transmitted via the IrDA stack). These are also explained in more detail in the following sections.

The other optional protocols are:

- IrCOMM - provides COM (serial and parallel) port emulation for legacy COM applications, printing and modem devices.
- IrOBEX - provides object exchange services similar to HTTP.
- IrTran-P - provides an image exchange protocol used in Digital Image capture devices/cameras.
- IrMC - specifications on how mobile telephony and communication devices can exchange information. This includes phonebook, calendar, and message data.

Recently further protocols have been added, more information can be found on the IrDA homepage.

Applications have the choice to use IrLMP directly, to use IrLMP through IrTTP, or to use one of the high level protocols built on top of IrTTP.

8.3 The IrDA Protocols in more Detail

8.3.1 IrDA Service Definitions

The IrDA protocols are defined in terms of services that are provided to higher layers. These services in turn are specified by service primitives and parameters. Only the service is specified in an abstract way, rather than the means by which a service is provided. This type of definition makes the specification independent of any particular interface implementation.

The following service primitives are used in the IrDA protocols:

Request	Passed from an upper layer to a lower layer to invoke a service.
Indication	Passed from a lower layer to an upper layer to indicate an event or to notify the upper layer of an action initiated by the lower layer.
Response	Passed from an upper layer to a lower layer to acknowledge some procedure invoked by an indication primitive.
Confirm	Passed from a lower layer to an upper layer to convey the results of a previous service request.

Figure 24 shows an example for IrLAP.

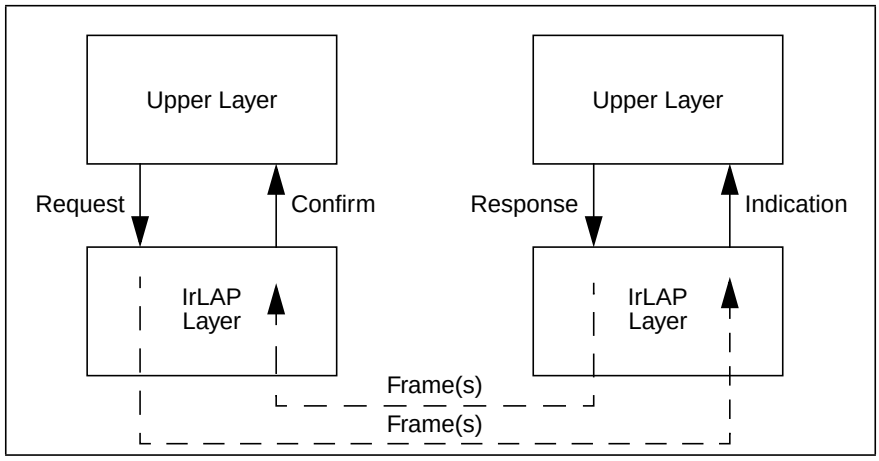


Figure 24. IrDA Service Primitives [12, p. 14]

8.3.2 IrPHY - The Physical Layer

The basic information about the physical layer, especially modulation and data rates, has already been covered in chapter 5. Therefore I won't go into it further here. The detailed specification is given in [11].

8.3.3 IrLAP - The Link Access Protocol

The full specification of IrLAP, including detailed specification of the state machine is given in [12].

The tasks of IrLAP are to do device discovery and to provide a device-to-device connection for the reliable transmission of data. The general procedure is as follows:

If a station wants to connect to another station IrLMP requests IrLAP to starts a discovery process. During this process all listening devices are requested to answer in response to (XID) discovery frames. The result is a list of available stations, including the (randomly chosen) device address for each station. If two or more stations happen to have chosen the same address the station doing the discovery has to execute an address conflict resolution mechanism. Then the concerned stations have to chose a new address. After discovery the 'connect'-service is used to establish a new connection between two stations. The station which initiated the discovery will become the primary station and control the connection. During setup various parameters such as: baud rate, maximum frame size, link

turnaround times, etc. are negotiated. After the connection has been established data can be transmitted in either reliable or unreliable mode. Disconnect closes a connection, if a station doesn't answer for some time (e.g. because it has moved) the connection is closed automatically.

IrLAP is event-driven, events are caused by requesting services (at the upper layer boundary), arrival of data (at the lower layer boundary) or the timeout of one of the timers. These timers are used to detect link activity (media sense), to signal that a station has used up its time quota and has to turn around the link, retransmission timeouts, etc.

8.3.3.1 IrLAP Service Definitions

To implement this functionality, IrLAP provides two general types of services ([12], chapter 2):

- Connectionless Services
- Connection-orientated Services

The Connectionless Services are as follows:

- Discovery Services: Find devices within communication range.
- Address Conflict Services: Resolve address conflicts following a discovery operation by causing the conflicting devices to select new (non-conflicting) device addresses.
- Unit Data Services: Transmit data outside of a connection. All data is broadcast and sent unreliable.

Connection Orientated Services:

- Connect Services: Establish a connection to a previously discovered station.
- Sniffing Services: Initiate a special low power connect procedure (sniffing).
- Data Services: Send data as either reliable, sequenced data (includes retransmission if necessary) or as unreliable, expedited, unsequenced data.
- Status Services: Inform the upper layer of bad link quality and a likely disconnection if the link quality doesn't improve soon. The upper layer is informed about unacknowledged sent data.
- Reset Services: Cause all unacknowledged data units to be discarded and all counters and timers to be reset. Only occurs if both ends of the connection agree.
- Disconnect Services: Terminate a logical connection and discard all outstanding data units.

8.3.3.2 IrLAP Frame Structure

Each IrLAP frame has the following format:

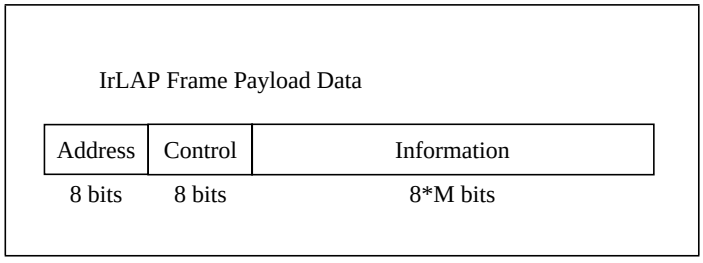


Figure 25. IrLAP frame format [12, p. 21]

- An address (A) field that identifies a secondary station connection address
- A control (C) field that specifies the function of the particular frame
- An optional information (I) field that contains the information data

Each of these fields is a multiple of eight bits, together they are referred to as the payload data. Each IrLAP frame is framed by fields which constitute a wrapping layer. The wrapping layer is a property of the physical layer, its use is to mark the beginning and the end of frames and to be able to detect errors during transmission. The fields depend on the particular physical layer being used, but in any case at least the components: start flag, frame check sequence, and stop flag must be included.

The control field defines the function of the frame as being:

Unnumbered format (U)	to establish and disconnect the data link, report procedural errors, transfer unsequenced data. These frames are used for data link management.
Supervisory Format (S)	these frames do not carry user data, but are used to acknowledge received frames, convey ready or busy conditions and to report frame sequencing errors.
Information Transfer Format (I)	These frames transport user data in the I field. Besides indicating the frame format, here the control field (C) contains send and receive counts that are used to ensure reliable (i.e. error free in-order delivery) communication.

[12, chapter 5] provides detailed information about the different types of frames.

Only one station at a time can transmit, the primary station has to release the link, then the secondary can transmit for a certain maximum amount of time after which it has to release the link back to the primary.

8.3.3.3 IrLAP State Diagram

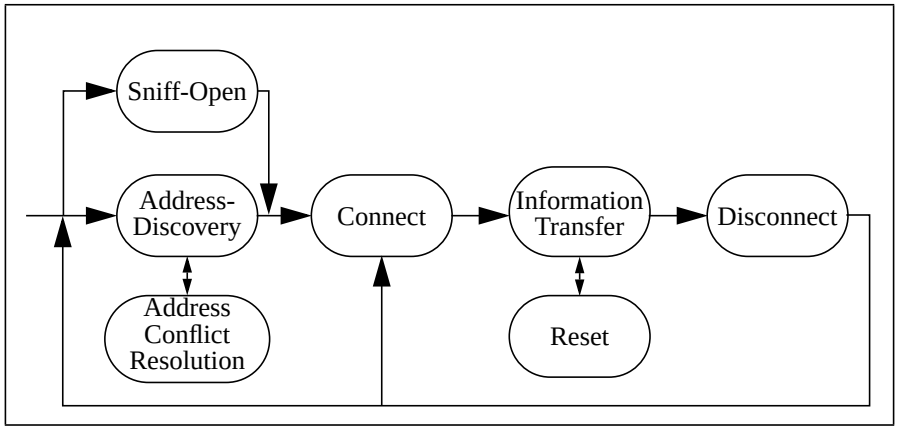


Figure 26. IrLAP state diagram [12, p. 34]

8.3.4 IrLMP - The Link Management Protocol

8.3.4.1 Architectural Components

IrLMP is specified in detail in [13]. It consists of two main components (see Figure 23):

- Information Access Service (IAS): Each IrDA device has to maintain an IAS which is an information base that allows other IrDA devices to query the local device to find out what are the provided services and how to access them. The information base consists of a number of objects holding the information, a dedicated protocol, and a set of operations to retrieve the stored information.
- Link Management Multiplexer (LM-MUX): The LM-MUX provides services to the local LM-IAS entity, to transport entities (like IrTTP) and to applications that directly bind to LM-MUX. While IrLAP provides one (physical) reliable connection between a pair of IrDA devices, the LM-MUX provides multiple (logical) data link connections over one IrLAP-link.

8.3.4.2 The Link Model

The LM-MUX provides services to carry out the Link Management itself and to its service users at Link Service Access Points (LSAPs). The LM-MUX can be in one of two modes:

When in **multiplexed mode**, several LSAP connections may actively use the underlying IrLAP connection. LM-MUX relieves the client entities of the requirement to coordinate access to the single IrLAP connection. However, it does not provide a per LSAP-connection flow control, which can cause deadlocks under certain circumstances. These problems are solved by IrTTP (see section 8.3.5.)

When in **exclusive mode**, just one LSAP-connection may be active on the IrLAP connection. This might be required by some applications if special control is necessary to achieve a reduced latency or control the link turnaround during their use of the link.

8.3.4.3 The Link Management Multiplexer

Link Management provides the following external interfaces at the upper and lower LM-MUX service boundaries ([13], section 3.1.1):

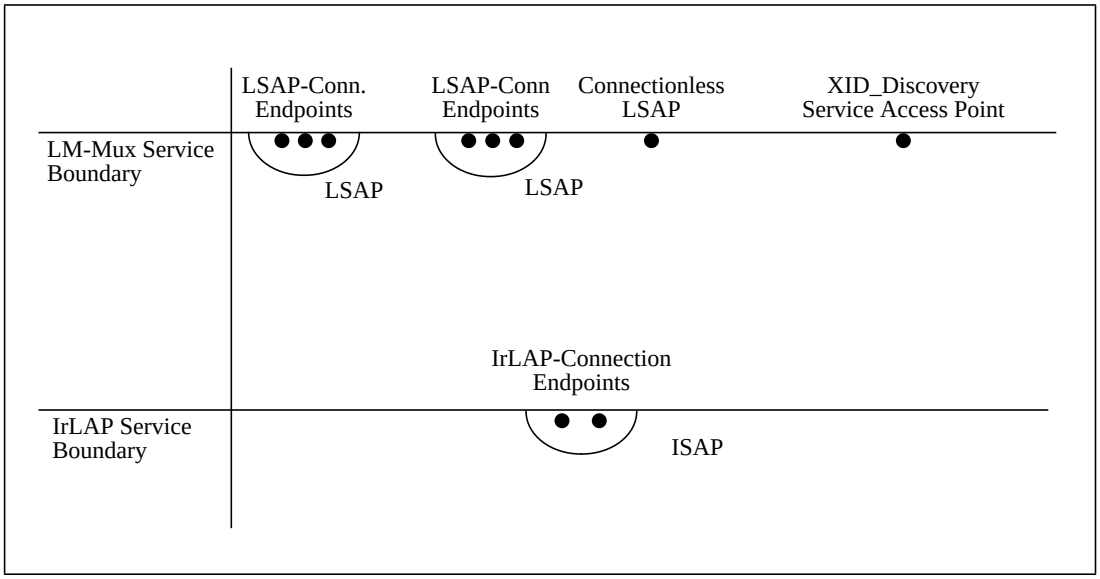


Figure 27. LM-MUX External Interfaces [13, p. 17]

- LM_Connect, LM_Disconnect, LM_Data and LM_UData service primitives are invoked at LSAP-connection endpoints. These are grouped together at an LSAP.
- LM_ConnectionlessData primitives are invoked at the Connectionless LSAP. LM_ConnectionlessData.indication primitives are delivered to all LM-MUX clients that bind to the Connectionless LSAP.
- LM_Discover and LM_Sniff primitives are invoked at the XID_Discovery Service Access Point.
- All IrLAP service primitives are invoked at an IrLAP-connection endpoint. There is one IrLAP Service Access Point (ISAP) per station.

Within a station LSAPs are distinguished by the value of the LSAP-SEL. The LSAP-values for both ends of an LSAP-connection are carried in the IrLMP header, thus a packet can be delivered to its correct destination.

The internal organization of the LM-MUX is shown in Figure 28:

Per station there is one Receive Demultiplexer, and one Station Control entity; while an LSAP-Connection Control Finite State Machine (FSM) is associated with each LSAP-connection endpoint.

The LSAP-Connection Control FSM is responsible for connection and disconnection of a single LSAP-connection endpoint with a peer LSAP-connection endpoint. During LSAP-connection establishment it

requests the use of a suitable IrLAP link from station control. Once the IrLAP connection is available, the FSM will try to contact the peer FSM to request a connection. If the peer FSM accepts, the connection will be established and data can be exchanged between the LM-MUX clients.

The Receive Demultiplexer is responsible for routing all IrLAP_Data.indication and IrLAP_Unitdata.indication primitives that arrive via any of the (possibly multiple, in case there are multiple physical devices) IrLAP-connection endpoints. The Receive Demultiplexer is also responsible for selecting a new LSAP-connection endpoint for the delivery of incoming Connect requests.

Station control is responsible for the transmission of connectionless data (invoked at the Connectionless LSAP), the operation of the XID_Discovery process and the associated IrLAP device address resolution, the connection and disconnection of IrLAP connections, the assignment of LSAP-connections to IrLAP-connections and the transitions between the Exclusive and Multiplexed LM-MUX modes.

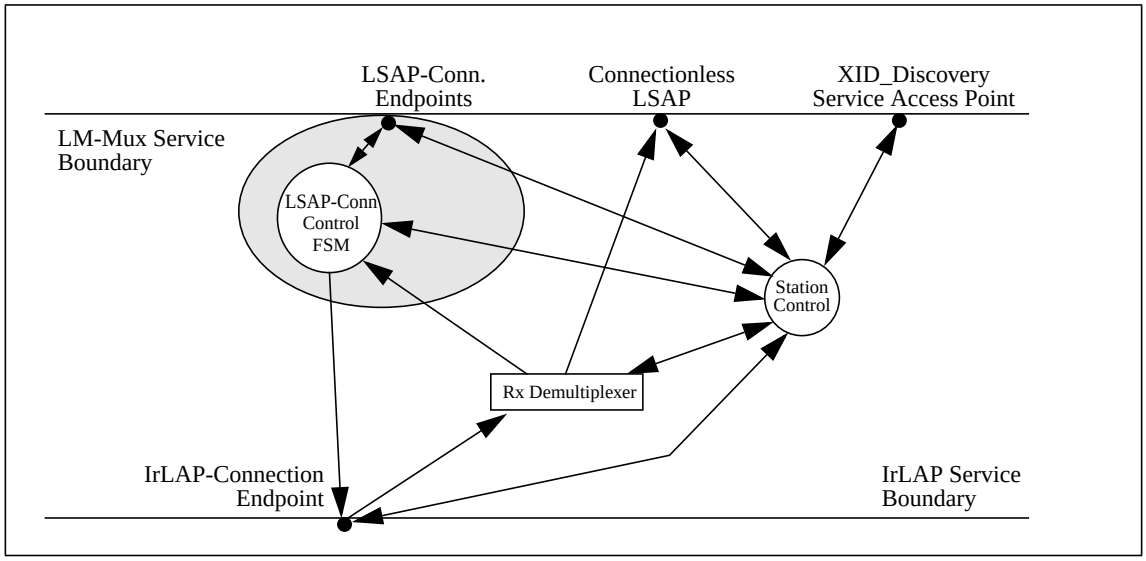


Figure 28. Internal Multiplexer Organization [13, p.19]

Note: LSAP-Sel 0 is reserved for the IAS-Server.

8.3.4.4 Frame Format

The IrLMP specification refers to IrLMP frames as LM-PDUs (LM-Protocol Data Unit). These are sent as (reliable) IrLAP data frames. Within the IrLAP data frame IrLMP uses a two byte header to encode the destination LSAP-selector and the source LSAP-selector. These uniquely identify an LSAP connection. For IrLMP data frames the data is sent directly following the IrLMP header. For Link Control Frames the IrLMP header is followed by an opcode byte and parameters (whose format depends on the type of Link Control Frame as specified by the opcode byte).

The Link Management Multiplexer is specified in full detail in [13, chapter 3].

8.3.4.5 Information Access Service

This part of the IrLMP protocol is specified in detail in [13, chapters 4 and 5].

When two IrDA devices establish a connection they have to follow the standard with respect to IrLAP and IrLMP. But normally they have no a-priori knowledge about any services that might be running on top of IrLMP as these services are optional. To solve this problem each IrDA compliant device needs to provide an Information Access Service (IAS).

The IAS service in a device contains information about the services provided by this IrDA device (to a remote device) and also provides operations for local service users to query a remote IAS on another device. This allows a client to find out the information that is necessary to use a particular service on a remote device. An example of such configuration information is the destination LSAP-selector to which to establish a connection in order to use a particular service.

Figure 29 shows the organization of IAS components. A Client IAP FSM is accessible through the Information Access Service Interface, this allows a local service to query a remote IAS. The Server IAP FSM accesses the local Information Base to handle remote requests.

Communication between a Client IAP FSM and a Server IAP FSM is defined through the Information Access Protocol which is a command/response protocol. Similar to the well-known-ports in the TCP/UDP protocols the IAS server is always located at LSAP selector zero. This is the only fixed LSAP selector. By this means a service on any IrDA device can always contact a remote IAS and then, at runtime, query for the necessary information in order to use an offered service. IAP defines one mandatory service primitive (GetValueByClass which returns a list of values of a given attribute of all objects of a given class), a number of optional service primitives, and the frame format used to transport the information.

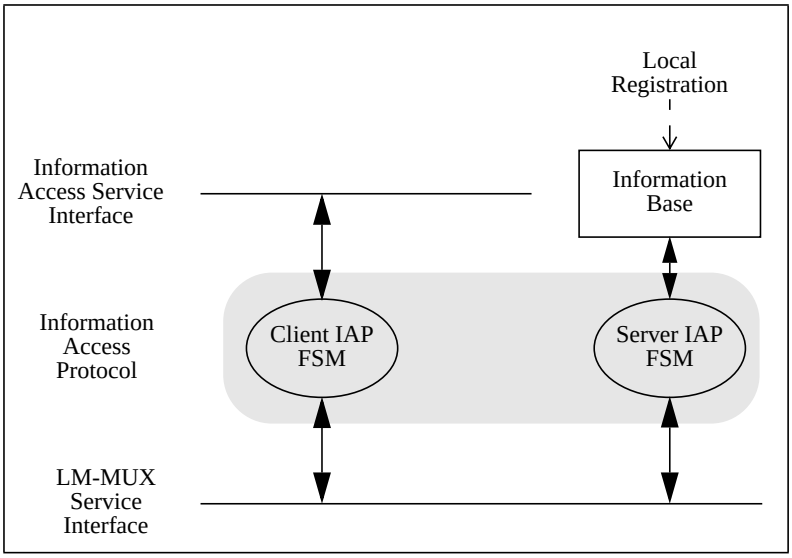


Figure 29. Internal Organization of the Information Access Service [13, p. 67]

Logically the IAS maintains a collection of objects and defines a standardized set of operations to query these objects. Each object in the database belongs to a certain object class, identified by the class name, and has a unique identifier, as multiple objects of the same class can exist. Information in the objects is maintained in terms of attributes which consist of a name and a typed value. The IAS supports a fixed set of base types, but no compound types. Values can only be written locally (local registration), only reading of remote values is supported.

An object of class “Device”, identifier 0, always must be present. It contains basic information about the device, such as its name, which of the IAS service primitives are supported, and possibly other information.

8.3.5 IrTTP - A Flow-Control Mechanism for Use with IrLMP

IrTTP is specified in [14]. Its goal is to provide flow control in combination with IrLMP. IrLAP provides flow-control between IrLAP entities. Adding IrLMP on top of IrLAP can introduce problems: “Reliance on IrLAP to provide flow-control for a multiplexed channel can result in dead-locks if the consumption of data from one multiplexed channel is dependant on data flowing in an adjacent multiplexed channel. Conversely, if inbound data on a multiplexed channel cannot be consumed and the underlying IrLAP connection cannot be flow-controlled off due to the possibility of deadlock, inbound data (freshly arrived or buffered) must be discarded in the event of buffer exhaustion. Sadly this reduces the reliable delivery service provided by IrLAP to a best effort delivery service provided by IrLMP LM-MUX (when multiple multiplexed channels are in operation).”([14], p. 1)

Possible solutions are to have each application provide its own flow-control mechanism on top of LM-MUX to ensure that there is always enough buffer space available for arriving data or to have each application provide a retransmission mechanism on top of LM-MUX that is able to recover from the loss of data in the case of buffer overflow.

The transport protocol TinyTP provides independently flow controlled transport connections, along with segmentation and reassembly.

8.3.5.1 Tiny TP Service Primitives

Each Tiny TP Service Access Point (TTPSAP) is accessible through one and only one IrLMP LM-MUX LSAP. Tiny TP provides the following service primitives [14, p. 2 ff.]:

TTP_Connect	establish a TTP-connection between two IrLMP LSAPs
TTP_Disconnect	reject incoming TTP connections, request to terminate a TTP connection and indicate both normal and abnormal termination of a TTP connection
TTP_Data	transmit client data (TTP Client Service Data Unit - SDU) to the peer TTP client. The service primitive TTP_LocalFlow is used to suspend and resume the generation of the TTP_Data.indication primitives.
TTP_UData	send data unreliably and without flow control, data is not guaranteed to be delivered. The primitives are directly mapped to the corresponding LM-UData primitives.
TTP_LocalFlow	control the flow of received TTP-SDUs between the receiving TTP entity and its local service user.

8.3.5.2 IrTTP Frame Format

IrTTP uses a one byte header, followed by a field containing the user data to transmit. The header byte contains a bit to signal if the user data field contains the last segment of a segmented TTP-SDU or if there are more segments to follow, and a 7 bit integer announcing a credit. The credit specifies the number of data-carrying Data TTP-PDUs that are allowed to be sent in the reverse direction.

Two types of TTP-PDU (TTP protocol data unit, i.e. header plus SDU) exist:

Connect TTP-PDUs are used during connection establishment and are carried in the UserData field of LM_MUX Connect LM-PDUSs. The credit field means “InitialCredit” here, i.e. the initial number of data-carrying Data TTP-PDUs that may be sent in the reverse direction. The user data field contains a parameter field which at the moment has a maximum size of 7 bytes. It is composed of two subfields, the first consisting of one byte, that specifies the size in bytes of the second subfield. This field contains a list of 3-tuples (PI, PL, PV) where PI specifies the parameter being carried, PL its length and PV contains the actual value. PI and PL are both a single byte in size, the interpretation of PV depends on the type of parameter as identified by PI. The rest of the user data field can contain up to 52 additional bytes of user data.

Data TTP-PDUs are carried in the UserData field of LM_MUX Data LM-PDUs. They are used for transporting client data to a peer TTP entity. In Data TTP-PDUs the credit field has the meaning of “DeltaCredit”, i.e. the number of additional Data TTP-PDUs that may be sent in the reverse direction.

8.3.5.3 Operation

Figure 30 shows the used variables at one end of a TTP connection that has reached its data phase.

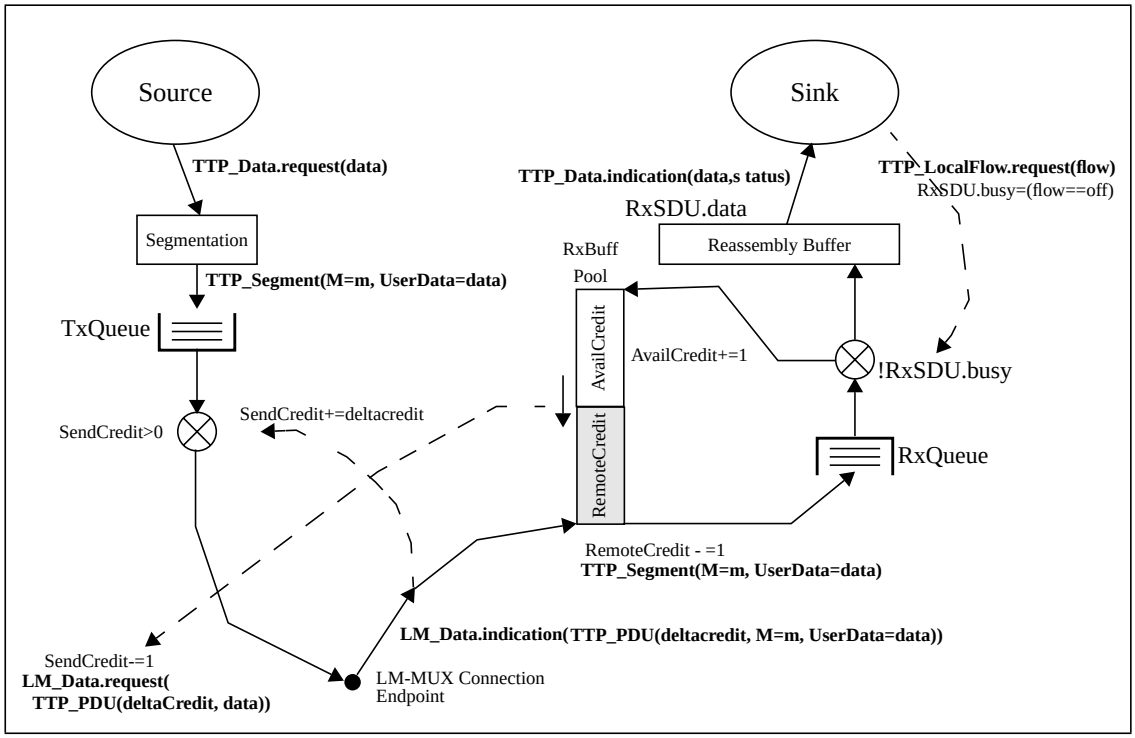


Figure 30. Tiny TP SAR and Credit Flow [14, p. 9]

The operation is as follows:

When sending the client (source) invokes `TTP_Data.request`. TTP, segments the data block if necessary, and puts the resulting segment(s) into the transmission queue `TxQueue`. If `SendCredit>0,` then the current value of `RemoteCredit` is inserted into the TTP_PDU as `deltaCredit` and `LM_Data.request` is invoked to transmit the generated TTP_PDU, `SendCredit` is decremented by 1. This procedure is repeated while there are segments waiting in the transmission queue and `SendCredit` is not zero.

If data is received, IrLMP invokes `LM_Data.indication` to deliver the data to IrTTP. The value of `deltacredit` is read and added to the current value of `SendCredit`. This allows IrTTP to start transmission again, if segments were waiting in the transmission queue. The data portion is put into the receiver queue `RxQueue` and `RemoteCredit` is decremented by one as one more entry in the receiver queue has been filled.

The client receiving the data frames can do flow control by invoking `TTP_LocalFlow` if it is in danger of running out of buffers, thus it can stop additional frames from being sent. Otherwise the segments waiting in the receiver queue will be passed on to the reassembly buffer. When a frame has been reassembled, IrTTP invokes `TTP_Data.indication` to deliver the frame to the client. For each segment being passed on from the receive queue to the reassembly buffer (thus freeing space in the queue) `AvailCredit` is incremented by one.

If `SendCredit` is non-zero the local TTP entity may send data which is guaranteed to be accepted by the peer TTP. If `SendCredit` becomes zero, then the local entity keeps the segments queued until the remote entity has new buffer space available which is indicated by a non-zero value of the received `deltacredit`.

If `RemoteCredit` is non-zero, then the remote TTP entity is allowed to send reliable data as there is free local bufferspace.

If `AvailCredit` is non-zero, then the local TTP entity has credit available that the remote entity doesn't know of yet. By setting `deltaCredit` appropriately in the next frame that is transmitted the credit can be passed on to the remote peer.

If `RemoteCredit` reaches some configured low threshold and at the same time `AvailCredit` is non-zero but the transmission queue is empty or `SendCredit` is zero (both cases keep the local entity from sending Data PDUs with an update to `deltaCredit`), then the available credit can be passed on to the remote TTP entity by using a dataless FlowData TTP-PDU.

8.3.6 LAN Access Extensions for Link Management Protocol - IrLAN

The IrDA protocols have reached wide support in industry and with the extension to FIR-mode at 4Mb/s the link is fast enough to be used as a network interface. IrLAN is a protocol built on top of the IrDA stack to support the following three modes:

- A device with an IrDA interface can access a local area network (LAN) through an infrared access point which is attached to the network (access point mode).
- Two devices with IrDA interfaces can communicate with each other as if they were attached to a LAN (peer mode).
- A device with an IrDA interface can access a LAN through a second device which is directly attached to a network and also is equipped with an IrDA interface (hosted mode).

One of the explicit design goals was that the protocol should be implemented as a set of network media-level drivers, i.e. the protocol runs as a device driver, no modification of existing (higher layer) network protocols should be necessary. IrLAN is specified in [15].

8.3.6.1 Overview

IrLAN uses two channels between a protocol client and a protocol server, also called provider. The provider is passive, it is the client's task to detect any provider and initiate a connection[1]. Typically the whole IrDA stack is initialized and IrLAN registers as a client. IrLMP causes IrLAP to do device discovery and if a remote device is discovered an appropriate indication is passed on to IrLAN. IrLAN requests a query via IAS for the IrLAN object in the remote information base. If this query is successful, i.e. the object is found which means that there is an IrLAN provider, then IrLAN takes the value read as an LSAP for the control channel and tries to open a connection to this LSAP which, if successful will become the IrLAN control channel. Through this control channel various link configuration parameters are negotiated and finally a data channel is opened. After that network packets can be sent and received via the data channel.

8.3.6.2 Frame Formats

IrLAN uses two frame formats, one for the control channel and one for the data channel. It supports 802.3 ethernet and 802.5 token ring, the data frame formats are the same as those for the native protocols on the software level. For ethernet this means the standard 14 byte header plus the data portion, while the preamble and FCS are omitted. Once the data channel has been established all network bound traffic (inbound and outbound) will go through this channel.

The control channel uses a client-initiated command/response protocol, each (client) request is immediately followed by a (provider) response. The control channel format is as follows:

Each request packet consists of three subfields, a one byte Command Code, a one byte Parameter Count and an up to 1020 bytes long Parameter List. A response packet has a similar structure, the Command Code is replaced by a Result Code, the other two fields remain the same. The Parameter List is a list of quadruples (Name Length[1], Parameter Name[1..255], Value Length[2], Value[0..1016]) of length - zero upto Parameter Count. As can be deduced from this description the protocol is clear-text based with Name Length and Parameter Name identifying the particular parameter and Value Length and Value giving its value.

1. Note that this is the reverse of what is desirable with respect to minimizing power consumption of the client. To minimize power it would be preferable if access points attempted to discover clients in their vicinity.

8.3.6.3 Commands

IrLAN defines the following commands, the sequence of their invocation is defined by the IrLAN state machine (see [15, p. 37 ff.]):

Command Code	Command	Description
0	Get Provider Information	Used by the client to determine the media type/data frame formats and the IrLAN modes supported by the provider (access point, peer-to-peer, and/or hosted)
1	Get Media Characteristics	Used by the client to get detailed information about the media types supported by the provider
2	Open Data Channel	Used by the client to get an IrLMP LSAP number on which it should establish a TTP connection to the provider for the data channel
3	Close Data Channel	When this command is received by the provider, it will stop sending packets to the data channel and will also stop sending received packets on the LAN. It is still up to the client to close the TTP connection.
4	Reconnect Data Channel	Used by the client to reconnect a data channel. If the reconnection is successful (the provider returns a status of zero), the state of the data channel is the same as when the channel was disconnected
5	Filter Configuration	Used by the client to control the filtering of packets from the provider to the client. This command also allows the client to check the filter configuration on the provider. Unicast and broadcast filter and a multicast filter list can be configured.

Table 25. IrLAN Commands [15, p. 16]

Via the Filter Configuration Command the client can request a unicast address from the provider which it can subsequently use as its MAC address. Thus an access point typically has two MAC addresses, one for itself, and one which it loans to the device at the other end of the IR link.

9. IrDA Implementation

This chapter describes my port of the Linux-IrDA-Stack to the SmartBadge. After a short introduction and an overview of the file layout I describe the changes that were necessary throughout the whole source code to port from Linux to the Badge. This is followed by sections which describe in more detail the changes necessary at the device driver level as well as at the upper level to integrate a UDP/IP stack. These sections are of special interest if the stack is to be ported to an operating system other than Angel. Finally I explain how to use the stack and list known problems. The listing of a debug log is contained in Appendix C.4.

9.1 Introduction

As network access for notebooks using the HP NetBeamIR-access points and an IrDA-network driver have proven to work well and the Badge is equipped with an IrDA-compliant infrared transceiver, capable of both the SIR- and the FIR-mode, Prof. Maguire suggested implementing IrDA on the Badge to give it wireless network access, which, as previously mentioned, would greatly extend the range of possible applications and services.

After reading the relevant standards it turned out that in order to use IrLAN the complete IrDA stack - IrLAP+asynchronous wrapper layer, IrLMP, IrTTP and IrLAN - have to be implemented. Further studies showed, that the state machines, especially those in IrLAP and IrLMP, are quite large. As I suspected that the available time would be too short to implement the whole stack from scratch I looked for alternatives and found an implementation in the freely available Linux IrDA kernel driver module, written by Dag Brattli, University of Tromsø, Norway. Initially my intention was to just get some ideas and to use small parts of the code. However, it turned out that the various files and layers were heavily interconnected so in the end finally I decided to try to port the whole stack. After the code to understand its structure and to find the “entry points” and the “red line through the code” I began by porting the lower layers, i.e. the device driver and the files implementing IrLAP. This was the hardest part as I had to figure out which parts of the complex Linux structures I could safely omit and which were necessary for the core functionality of the protocols, as well as how to imitate special Linux kernel functionality through the use of functions that are available under Angel. As the IrLAP device discovery procedure is the first step in establishing an IrLAN link my first task became to make this part work properly. In the course of implementing/porting this first part I had to solve nearly all of the problems that were a result of the different underlying operating system architectures. The solutions I developed there subsequently proved to be usable with the higher layers. Thus, after solving these initial problems and porting the higher layers (which already worked under Linux) the resulting stack worked as expected. Despite this, a few subtleties remained hidden and required many hours of intensive debugging. The details of this whole process are described in the subsequent sections.

9.2 File Layout

In this section I give a short overview of the source file structure of the IrDA-protocol stack. Although I took all the files of the Linux IrDA stack and ported them to the Badge file by file some additional files needed to be added to contain functions and declarations that are expected to be provided by the Linux kernel. Under Linux `irda_device.c` provides an abstract device driver interface and `irport.c` implements one specific driver (SIR driver for serial port). However, on the Badge the serial port associated with the IR transceiver is the only possible device, therefore the functions of these two files could safely be combined. Hence in a future release the files `irda_device.c` and `irport.c` could be merged.

In the following list the files are grouped by functionality:

General support code (provides Linux kernel functionality not available under Angel)

byteorder.s	functions to convert between little and big endian byte order
byteorder.h	
skbuff.c	functions to manipulate elements of type sk_buff (central type to handle network packets)
skbuff.h	
errno.h	Linux error codes
coretimer.h	Linux timer types and definitions (originally file /usr/include/linux/timer.h)
lock.h	implementation of the Linux lock functionality
spinlock.h	Linux lock functionality declarations
types.h	Linux general type declarations
unaligned.h	implement unaligned memory access

General IrDA related functionality

irda_timer.c	timer functionality (general start_timer function, timeout callback functions)
timer.h	timer functionality (declarations, timeout values, specific start timer functions as inlines)
irmod.c	IrDA stack initialization and cleanup, only a few functions are left, these could be added somewhere else
irmod.h	
irqueue.c	general queue and hashbin implementation
irqueue.h	
irda.h	(originally file /usr/include/net/irda/irda.h) general IrDA types and definitions
linux_irda.h	(originally file /usr/include/linux/irda.h) general IrDA types and definitions

Device driver functionality

irda_device.c	abstract device driver interface, could be merged with `irport.c`
irda_device.h	
irport.c	serial port device driver
irport.h	
netdevice.h	declares `struct device` used in `irport.c`
crc.c	SIR-mode CRC code
crc.h	
wrapper.c	IrDA SIR async wrapper layer
wrapper.h	

IrLAP layer

irlap.c	general IrLAP functionality
irlap.h	
irlap_event.c	IrLAP state machine implementation
irlap_event.h	
irlap_frame.c	Handling of IrLAP frames
irlap_frame.h	
qos.c	Quality of Service negotiation for IrLAP

qos.h

IrLMP LM-MUX

irlmp.c	General IrLMP functionality
irlmp.h	
irlmp_event.c	IrLMP state machines
irlmp_event.h	
irlmp_frame.c	Handling of IrLMP frames
irlmp_frame.h	
discovery.c	Handling of discoveries at the IrLMP layer
discovery.h	

IrLMP IAS

irias_object.c	Implementation of the IAS information base
irias_object.h	
iriap.c	IAP protocol implementation
iriap.h	
iriap_event.c	IAP state machine
iriap_event.h	

IrTTP

irttp.c	IrTTP implementation
irttp.h	

IrLAN

irlan_common.c	general code that is common to both the provider and the client
irlan_common.h	
irlan_event.c	set new state for client and provider state machine
irlan_event.h	
irlan_client.c	implementation of the IrLAN client
irlan_client.h	
irlan_client_event.c	client state machine
irlan_provider.c	implementation of the IrLAN provider
irlan_provider.h	
irlan_provider_event.c	provider state machine
irlan_filter.c	handle filter requests to the provider
irlan_filter.h	
irlan_eth.c	ethernet network device driver interface
irlan_eth.h	
if_ether.h	declares ethernet constants used in `irlan_eth.c`

IP/UDP stack (not part of the IrDA stack, Angel with IP stack compiled in is required)

ipstart.c	initialize the IP/UDP stack and get an IP address via BOOTP
address.h	ARM header defining address types used in the IP/UDP stack

ip.h	ARM IP stack types and definitions
udp.h	ARM UDP stack types and definitions
socket.h	ARM socket types and definitions

9.3 Changes Applied to the whole Stack, Problems in Porting

I will divide the changes to the code that were necessary to make the stack run on the SmartBadge into two sections:

- The first set of changes resulted from different compiler capabilities and some particular Linux kernel functions that needed to be provided. These changes had to be applied throughout the whole stack ,but now are implemented and shouldn't cause further problems if the stack is ported to another OS on the Badge. I'll discuss details of these aspects in the remainder of this section.
- The second set of changes applies to the low level interfaces. These changes mainly concern the device driver layer and the interface to operating system features such as memory management and especially interrupt handling, timers, scheduling, and locking mechanisms. If the stack is ported to another operating system these parts of the code are very likely to require adaptation again. These issues will be handled in the next section.

The first problems that I encountered was that the Linux code uses some features of the GNU C compiler that the ARM compiler does not support. Some of these are:

- Quite a number of enum declarations in the Linux code have a comma after the last defined type, e.g. they are of the form `typedef enum {a,b,c,} d;` which is not accepted by the ARM compiler and had to be changed for each occurrence into something of the form: `typedef enum {a,b,c} d;`
- Inline declarations of functions are contained in the C-files. The ARM compiler requires inline function declarations to be in header files. Therefore all inline functions had to be moved from the C-files to the corresponding header files. This caused new troubles as many of the inline functions were defined as static which was not possible when moving them to header files.
- The GNU compiler supports macro definitions with a variable number of arguments such as used in `printf`. This feature is used heavily for debug purposes. These macros can be used much like `printf`, but take an additional parameter that specifies the debug level, for example `DEBUG(2, "Debug value: %d\n", val)`. The preprocessor compares this first parameter with the value of a globally defined symbol `DEBUG_LEVEL` and generates code only if the value of this symbol is equal or greater than the given parameter. This allows the programmer to easily adjust the amount of debug code that is generated and later executed. Usually a value in the range 0..4 is used as an argument to the macro. The ARM compiler does not support a variable number of arguments in macros, therefore this elegant method could not be used and instead all occurrences of `DEBUG` with a variable number of arguments had to be commented out. Using preprocessor commands this feature was simulated for one argument such as in the example given above. The macro is called `DEBUG_1` and defined in `utillib/include/util_debug.h`. Using this method at least single variable values could be output during debugging. Also the GNU compiler predefines a symbol __FUNCTION__ which contains the name of the current function. This symbol is used in every call of the `DEBUG` macro to clarify the origin of the debug messages. Unfortunately this symbol was not supported by the ARM compiler and therefore the name of the function had to be added to all the debugging messages manually.
- Throughout the code packets have to be handled. As the Linux kernel module implements a network device driver which higher network layers can use to transmit and receive network packets, the code uses the Linux networking specific `sk_buff` structures and functions to handle all packet operations. The data field holding the actual data to be transmitted is a `char` pointer to a memory block. To facilitate access to the different fields of a packet, for each IrDA packet type a structure is defined. The structures usually contain `char`, `short` and `int` members to describe the packet fields. In order to access the packet fields the `sk_buff` data pointer is simply cast to a pointer of the appropriate packet type, which then allows convenient manipulation of the fields. The problem now is that in many cases a fixed set of header or parameter fields is followed by a variable length data field. The GNU

compiler allows declarations of the form "`char data[0]`". This effectively defines a pointer of type `char` which is automatically initialized to point to the proper location in memory. Unfortunately this elegant method is not supported by the ARM compiler. After first spending some time to find understand the meaning of an "array of length 0" and then spending quite a while trying to find a reasonable workaround (which at the beginning regularly resulted in crashes) I tried to replace these statements simply by "`char data[1]`". This works because it defines a pointer to the current memory location and in (this case) it is fortunate that C does not check array boundaries, so given the above declaration, statements like "`data[5]`" work without problem. There is no danger of an invalid memory access as the first element points to the correct address and further accesses simply access memory space in the `sk_buff` data field.

- Another problem that required quite a long time to solve was that the ARM compiler delivered in the ARM SDK version 2.11, sometimes produces erroneous code: in (at least) two locations in the file `irlap_frame.c`, within the functions `irlap_send_discovery_xid_frame()` and `irlap_send_ua_response_ frame()`, integer assignments between a `char` pointer and a packed structure were not translated properly, but only if the variable being assigned had not been used before. I found the error only by chance when trying to find the reason for incorrect values in the discovery messages. After adding debug output to check the values before and after a function call it suddenly worked. After some additional investigation I finally realized that adding a statement of the form "a=a;" before the actual assignment prevented the error. The newer version of the compiler, delivered with the ARM SDK version 2.5 doesn't have this error any longer.

The changes described so far were primarily the result of incompatibilities between the Linux GNU compiler and the ARM compiler. In addition, I made some additional changes that are a result of the different requirements. As already mentioned, under Linux the IrDA stack is accessed as a device driver, it looks like a physical network interface to the higher layers. Any higher layer can bind to the stack and in turn the stack can bind to any available (physical) device driver, e.g. drivers for different chipsets, IR dongles, both in SIR mode (where the use of serial devices drivers easily allows you to run the IrDA stack via a serial cable) and in FIR mode. To support this functionality some additional information is necessary to enable the binding between the different layers. This information is mainly kept in the structures defining the various layers. In addition, in every packet passed as an element of type `sk_buf,` information about the sending and receiving device instance must be kept to allow proper routing in the network stack. On the SmartBadge this can be substantially simplified. There is only one instance of any type of device, therefore device pointer lists kept in the various structures and device pointers as parameters to some functions could be removed and simply be replaced with a single global variable of this type. These changes were applied in order to simplify the structure and to save memory. In this context I also removed the "`magic`" fields that are defined in many of the structures to make sure that a pointer being assigned really points to an instance of this type. Some more variables that turned out not to be needed in this simplified environment were removed. It might even be possible to remove further variables but I did not have the time to go into the code again and cross check which variables one could safely get rid of without disturbing the stack's operation.

A number of functions to manipulate `sk_buffs` could also be removed as their functionality is not needed on the SmartBadge. On the other hand some functions that are kernel functions in Linux had to be added for use with the SmartBadge under Angel. These are mainly concerned with memory management, scheduling, and locking mechanisms. As these might have to be adapted if the stack is ported to another OS they will be discussed in the next section, along with the low level device driver. For the higher layers, except for the receive and transmit functions in IrLAN which would also have to be adapted and are discussed in section 9.5, most of the code should be directly portable to other systems.

9.4 Changes at the Low-Level Interface, Porting to other Operating Systems

The necessary changes in the low-level parts of the IrDA stack took place in the following areas:

1. memory management
2. locking mechanisms
3. timers and scheduling
4. device driver (irport.c, wrapper.c)

9.4.1 Memory Management

As the Linux IrDA stack is implemented as a Linux kernel module it uses special kernel functions for memory management. The Linux kernel provides its own memory management. It manages a set of differently sized memory blocks in a cache, the memory allocation routine then chooses a suitable block and returns it. Having predefined sizes and the cache structure avoids fragmenting memory and provides high performance. As this kind of memory management was not available on the SmartBadge under Angel, I changed all relevant function calls (mainly calls to `kmalloc()`and `kfree()` (defined in /usr/src/linux/mm/slab.c) and some functions in `skbuff.c` that directly accessed the memory pool) to use the standard library functions `malloc()` and `free()`. As far as memory management is concerned the ported version should compile and run under any operating system that provides the standard memory functions. In case an OS implements a different memory management it should be sufficient to simply change all calls of `malloc()` and `free()` to the appropriate function.

9.4.2 Locking Mechanisms

As Linux is a multitasking system it has to provide the possibility to lock certain regions of code/memory for exclusive use by one task and thus provides a number of mechanisms to achieve that. The kernel modules mainly use a set of macros that are defined in `spinlock.h`. They offer three debug levels, at level 0 (no debugging) most of them don't execute any actions, just a couple of them are defined to enable or disable interrupts and if necessary save/restore the processor state flags. At higher debug levels additional checks are carried out. I only used level 0 and thus it was enough to map the Linux functions `cli()` and `sti()` (clear interrupts and set interrupts) to the corresponding Angel functions `Angel_EnterSVC()` and `Angel_ExitToUSR()`. They switch the processor mode to CPU Supervisor Mode and back to User Mode. The functions are accessed using the mechanisms explained in section 4.2. As these functions not only disable/enable the interrupts, but instead perform a complete mode switch no processor flags need to be save/restored explicitly. Another set of functions, defined in `lock.h`, allows atomic testing and setting/resetting of bits. This allows programs to lock a region for a longer time than would be possible by executing in Supervisor Mode as this disables all interrupts. These functions are also based on `Angel_EnterSVC()` and `Angel_ExitToUSR()`. Again for porting to another OS it would be sufficient to replace these two functions by the respective functions in the new OS. Depending on the implementation it might be necessary to also implement the functions `restore_flags(x)` and `save_flags(x)`.

9.4.3 Timers and Scheduling

The IrDA stack is mainly event-driven, besides service requests by clients the events are communication events (data arriving) and timer events. Each of the protocol layers uses a set of timers for different actions. The number of timers required for the IrDA protocol stack is many more than the four hardware timers available on the StrongArm processor. Therefore it is necessary to use software timers. Linux already provides these in the kernel, so I had to implement software timers that can use the Linux timer datatype (`struct timer_list`). The solution was presented in section 4.7.4. In order to port this code to another OS, the functions `ostimer_InitTimer()`, `ostimer_AddTimer()` and `ostimer_DelTimer()` have to be provided or the respective function calls have to be replaced by equivalent calls. Also an equivalent to the variable `ostimer_ticks` which keeps track of the time base of the software timers has to be provided.

Furthermore the bottom-half-handler mechanism as described in section 4.7.4 has to be implemented or replaced. All three currently implemented bottom-half-handlers are located in the file `irport.c`.

Note that `irlan_eth_flow_indication()` in `irlan_eth.c` contains code to request execution of a fourth bottom-half- handler when a device busy state has been cleared. The service is meant to schedule the higher layers to send any packets that were buffered as a result of the busy condition. The Fusion IP/UDP stack that is currently used doesn't query the busy state of the device before sending, so no handler is supplied at the moment. If this feature is to be used, then the stack has to query before sending and needs to buffer the frames if the device is busy. When the device becomes available again, execution of the bottom-half-handler is requested through `irlan_eth_flow_indication()`. The handler then has to notify the stack that it can now send the previously buffered frames.

9.4.4 Device Driver

In this section I describe the implementation of the device driver `irport.c` in more detail. Basically I took the existing Linux code and removed a few functions that turned out not to be relevant for the implementation on the SmartBadge and replaced the bodies of most of the remaining functions with my own code, i.e. I left the interface the same, but changed the internal implementation. The source code for the files `irport.c`, `wrapper.c,` and `wrapper.h` is listed in Appendix C.

The code contains `#ifdef SA1100` sections to differentiate between code for transmitting on the new Intel StrongARM and the original StrongARM which has the SIR-transmission bug. At the time of coding I was expecting SmartBadge 4 with the successor chip (SA1110) which doesn't have this bug. In the meantime I received a couple of SmartBadges equipped with the corrected Intel StrongARM. If defined `SA1100` means the original, buggy StrongARM, while `!SA1100` means the Intel StrongARM1100 or the StrongARM1110. The code for software modulation is still contained in the file as I hoped to be able to do at least the initial discovery and negotiation in software modulated SIR mode and then as soon as possible switch to FIR mode. But as described in section 5.2.2 it turned out that in a system with a high rate of interrupts it is difficult to properly do software modulation.

Line 122 defines a constant which is used in line 397 to get around the UART TBY problem as described in section 4.8.4.6.

In lines 161 ff. the "Quality of Service" structure is initialized with the available baud rates. It is used when setting up the connection to negotiate the final data rate with the peer device. When doing software modulation only 9600 baud and 4 Mbaud are available. When using normal UART transmission all defined baud rates between 9600 baud and 115200 baud for SIR as well as FIR will be supported in a final version. This version does not yet implement FIR mode, mainly because FIR only makes sense in combination with DMA data transfers, otherwise the interrupt load would probably be too high. I tried DMA with serial port 3, but did not succeed in making DMA work within a reasonable amount of time. Although I now know of source code for the SmartBadge that successfully uses DMA. Once it is determined how to configure the DMA controller, adding FIR mode for the IrDA stack should be relatively straight forward. I have already added place-holder code in the obvious locations, thus mainly an FIR-interrupt handler will have to be added. Since in FIR mode wrapping/unwrapping and CRC-checking are done in hardware the FIR handler can directly deliver received data to the upper layer, without having to invoke the unstuffing process, similarly transmit data can also be taken directly from the upper layer and transmitted without the need for executing the stuffing code on the frame before transmission.

Line 173 ff. initializes the transmit and receive queues that hold the already mentioned sk_buffs. The size for the device transmit and receive buffers is set, then two local buffers are allocated. I'll explain the function of these buffers shortly.

The following functions deal with opening, closing, and starting the device. I chose to keep the Linux terminology, although the names probably don't always really match the functionality in my implementation, but this allowed me to leave the interface the same and thus the higher layers did not have to be changed in terms of their interfacing to this code.

In line 299 the function irport_change_speed begins, it is responsible for programming UART 2 in order to operate at the selected baudrate. As already mentioned currently only SIR mode is supported so the function only needs to reprogram UART control register 2. Nevertheless it already contains all necessary code to deal with FIR as well, just as soon as an interrupt handler is available.

The next 2 functions toggle the device between transmit and receive mode. In case of irport_sir_set_transmit_mode and a new processor not only the transmitter is enabled but also the transmit interrupt. Thus the transmission immediately begins, i.e. the transmit buffer has to be set up before calling this function. For software modulation mode an additional function is necessary to actually start the transmission.

Following the two mode select functions the interrupt handler for software modulated transmission is defined. For more information see section 5.2.2.

The remaining functions are the core functions for sending and transmitting. I'll describe them in logical order, starting with transmission:

- If a packet has to be sent, `irlap_queue_xmit()` (in `irlap_frame.c`) enqueues the sk_buff holding the data into the transmit queue `tx_queue`. Following that it requests the transmit bottom handler service by calling `ostimer_MarkBH(OSTIMER_BH_IRPORT_TRANSMIT)`.
- The next time a timer interrupt occurs the request is detected and `irport_BHTransmit()` (line 840) is queued for execution. When this function is executed it first unmarks the bottom half handler request and then checks if there is an element available in the transmit queue. If the device is not busy, the sk_buff is dequeued and the `function irport_hard_xmit()` is called, the buffer is passed on as an argument.
- `irport_hard_xmit()` (line 603) marks the device as busy and then copies the packet data into the device transmit buffer `tx_buff`, using the function `async_wrap_skb` (in `wrapper.c`) to do the framing and stuffing. After that it calls `irport_sir_set_transmit_data()` which switches to transmission mode, enables the transmit interrupt and thus starts the transmission.
- The interrupt handler (line 649) detects that it is a transmit FIFO request and calls `irport_write_wakeup()` (line 553, another holdover from Linux, in order to save time by avoiding an additional function call this code could be integrated into the interrupt handler itself). The function is executed every time a transmit interrupt occurs and first checks, if there is any data left in the buffer. If so, it gets max(4, #bytes left in buffer) bytes and writes these directly into the FIFO, thus transmitting them. Then it polls the "transmitter not full" flag and writes as many additional bytes as possible. When at some time the buffer is empty it requests the transmit bottom handler service again and switches to receive mode.
- If there are packets left in the queue the process will start over, when the bottom half handler is executed the next time.

For receiving the process begins in the interrupt handler:

- When the receive FIFO has been filled to between one third and two thirds of its capacity, receive FIFO service is requested via the interrupt.
- The interrupt handler (line 649) reads as many bytes as are available and stores them in the local receive buffer `ir_rec_buf`. When it is finished it requests execution of the unwrap bottom half handler.
- `irport_BHUnwrap()` as usual clears the request and copies the data into the local buffer `ir_unwrap_buf`. This operation is executed in supervisor mode to guarantee that it can lock the buffer. After releasing the lock by returning to user mode, then for each byte the function `async_unwrap_char()` is called.
- `async_unwrap_char()` (`wrapper.h`) is a wrapper that calls the appropriate function of the unwrapping/unstuffing frame machine. The FSM functions copy each unstuffed data byte into the device receive buffer (`ir_device.rx_buf`).
- When a packet has been completed the appropriate state machine function `state_inside_frame()` calls `async_bump()` (also in `wrapper.c`) which allocates a new `sk_buff`, copies the received data (from `ir_device.rx_buf`) into the sk_buff's data buffer, and enqueues the new `sk_buff` into the receiver queue `rx_queue`, if this is not yet full. Than it requests irport receive bottom half handler service.
- `irport_BHReceive()` (line 872) unmarks the request and dequeues sk_buffs from the receive queue until this is empty. Each sk_buff is passed up to the IrLAP layer by calling `irlap_driver_rcv()`.

If the stack is to be ported to another system the following parts of irport potentially have to be changed, depending on the interface the OS offers:

- Interrupt handler
- Integration of the bottom half handlers
- possibly access to the serial port

Note: the copy from ir_rec_buf to ir_unwrap_buf in irport_BHUnwrap() is an addition to the Linux code. In Linux asynch_unwrap_char is directly called by the interrupt handler when it has a byte to process. On the SmartBadge this lead to problems, my suspicion was that the unstuffing/unwrapping for some reason took too long for an interrupt operation. I can't prove it, but after moving the unstuffing action into a callback these problems did not occur any more.

As already mentioned an FIR extension could directly pass the data on to `irlap_driver_rcv()` once a complete frame has been received.

9.5 High-Level Interface, Integration of the IP/UDP Stack

At the upper boundary of the IrDA stack two functions are involved in the integration with the IP stack: `irlan_eth_xmit()` and `irlan_eth_receive()`, both of them defined in the file `irlan_eth.c`.

In the IP stack the functions `irlan_device_read()` and `irlan_device_write()`, both located in `irlan_device.c` required some modification. The problem in combining these two stacks was that they use a completely different memory management:

The IrDA stack uses the normal clib memory functions, i.e. `malloc()` and `free()`. Using these functions, memory blocks of type `sk_buff` are allocated, passed through the different layers, used and changed and finally freed again. The most efficient way would have been to just pass these blocks on and up the IP stack as well. This was not feasible due to two main reasons:

- it would have caused a major rewrite of the stack to integrate these types into the existing structure
- the IP stack is compiled and linked with Angel, i.e. is part of the "kernel". Angel does not provide the usual malloc/free functions there - even if I had rewritten the functions in the IP stack to use the sk_buffs I could not have released old or allocated new buffers. Angel and also the IP stack use a large static memory block and implement their own memory management on top of that. This results in a model where the caller of a function, if he passes in a buffer is returned an equivalent (possibly different) memory block in order to keep the balance.

My solution is as follows:

- write operation: `irlan_device_write()` gets a buffer containing the data to transmit and has to return a balance buffer to compensate for the received buffer. The function now just assigns the data buffer to the balance buffer pointer. Then it calls `irlan_eth_xmit()` which through an adapted interface gets the buffer and its length. Within `irlan_eth_xmit()` a new `sk_buff` memory block is allocated and the data is copied into it. The `sk_buff` is then sent down the IrDA stack. `irlan_device_write()` returns and delivers a balance buffer which just happens to be the same one it got. When the IP stack is initialized a modified init function (originally `netstart_main()`, now called `Angel_NetstartMain()`) is passed a pointer to a structure containing the address of `irlan_eth_xmit()`, i.e. the mechanism used to call Angel functions from outside is now reversed (i.e., Angel is now able to call on user added functions within the kernel)
- read operation: here the case is a bit more difficult: `irlan_eth_receive()` is called when there is new data to deliver, i.e. it' s purely event-driven. The IP stack on the other hand has to be polled for receiving operation. `irlan_eth_receive()` takes the data out of the received `sk_buff` and copies it into an internal IP stack buffer whose address it was passed during the initialization -- this is in the same structure as the pointer to `irlan_eth_xmit()`. A further variable is set to the length of the received data. Then it calls the IP stack polling function `ethernet_process_one_packet()` (with parameters buffer and length,) which queries the lower layers (of the IP stack) if any packets have arrived. If so, then it processes them and passes them on to

the appropriate higher layer callback function. `ethernet_process_one_packet()` as being called has to return an equivalent buffer to the caller. `irlan_eth_mxit()` thus it passes in the buffer containing the new data and in return gets another buffer which it can use for the next call.

In summary, to integrate the IrDA stack with an IP stack in IrDA only the two top-layer functions irlan_eth_xmit() and irlan_eth_receive() have to be adapted. My solution was relatively easy to implement because it didn't require major changes, but is not a good solution in terms of performance as for each packet being sent and for each packet being received the buffer has to be copied in one direction.

The main problems in the porting process were to understand the Angel and IP stack buffer management and the fact that the IP stack is statically linked with Angel, thus for each small change in the code the whole Angel-image had to be rebuilt and downloaded to the FLASH - and - if there is a problem, Angel may not even boot anymore. While the IrDA stack debugging was not extremely easy, one could always get at least some information before the program crashed. However, if there is an error in Angel one often does not get any information, which makes the search for the error quite complicated and time consuming.

Apart from the necessary changes to make the two stacks interoperate I also changed some code in the modules `arp.c` and `bootp.c`. In the arp module in function `arp_resolve()` I replaced a simple counting loop to wait between retransmission by an exponential backoff timeout based on the OS timer count register. Similarly in the bootp module I replaced the ebsa110-specific timer code with an exponential backoff strategy again based on the StrongARM timer count register.

9.6 UDP over the IrDA-Stack

Here, using a small example application acting as a UDP echo server on the Badge, I describe how to use the IrDA stack together with the IP/UDP stack. First the main application is explained, then the function to initialize the IP stack is discussed.

9.6.1 UDP Echo Server on the Badge

The following listing shows all necessary steps to set up and start the stacks and to transmit and receive UDP packets via the socket interface:

```
 1 /*
 2  *      Description:
 3  *          Simple udp echo server demonstrating the use of UDP/IP over IrLAN
 4  *          via the socket interface.
 5  *
 6  *      The program initializes and starts both the IrDA stack and the IP
 7  *      stack, requests an IP address via bootp then, sends three sample
 8  *      UDP packets to a given destination address.
 9  *      Afterwards it enters a loop checking for UDP datagrams sent to port
10  *      7, the echo port. Datagrams are read, print to stdout and returned
11  *      to the sender.
12  *
13  *
14  *      --Christoph Wolf
15  *         chwolf@it.kth.se
16  *
17  */
18
19
20 #include <stdio.h>
21 #include <string.h>
22 #include <stdlib.h>
23 #include <irda/timer.h>
24 #include <irda/irmod.h>
25 #include <util/util_interrupt.h>
```

Listing 35. UDP echo server on the Badge

```
26 #include <util/util_serial.h>
27 #include <util/util_debug.h>
28
29 #include <irda/byteorder.h>
30 #include <irda/socket.h>
31 #include <irda/socket.h>
32
33 // the timer channel to be used for the software timers
34 #define LIST_TIMER_CHAN 1
35
36
37 int SendUDPPacket(int sock);
38 unsigned int inet_addr(unsigned char a, unsigned char b,
39                        unsigned char c,unsigned char d);
40 int ipstack_init(ip_addr client_ip, ip_addr cur_ip);
41
42 #define BUFLEN 512
43
44
45 int main(void)
46 {
47
48     // provide an ip address if bootp doesn't succede
49     ip_addr ip = {130,237,15,240};
50
51     int i;
52     int sock;
53     int len, ret;
54     struct sockaddr_in my_addr;
55     struct sockaddr    source_addr;
56     int source_len;
57     char buf[BUFLEN];
58
59     printf("begin\n");
60     fflush(stdout);
61     int_Init();                // initialize the interrupt module
62     debug_Init();              // initialize the debug module
63     debug_String("\n\n\n\n\n\rmain: init\n\r");
64     misc_InitAngelFunctions();  // initialize the function pointers to
65                                // Angel functions
66
67
68     OSTIMER_DISABLE_INT_ALL;    // disable all operating system timers
69     ostimer_InitListInt(LIST_TIMER_CHAN);   // initialize the software
70                                             // timer functionality
71
72
73     irda_init();               // start up the IrDA stack
74
75
76     if((ret=ipstack_init(ip, ip))!=0)   // initialize the IP stack.
77     {
78         printf("bootp not successful\n");
79
80     }
81
82
83     // open a socket
84     if (0 > (sock = Angel_Socket(AF_INET, SOCK_DGRAM, 0)))
85     {
86         return -1;
87     }
88
89     // send some sample UDP datagrams
90     i = SendUDPPacket(sock);
```

Listing 35. UDP echo server on the Badge

```
 91     printf("SendUDPPacket() returned %d\n", i);
 92     i = SendUDPPacket(sock);
 93     printf("SendUDPPacket() returned %d\n", i);
 94     i = SendUDPPacket(sock);
 95     printf("SendUDPPacket() returned %d\n", i);
 96     fflush(stdout);
 97
 98
 99     // bind to the echo port on the local address
100     my_addr.sin_family = AF_INET;
101     my_addr.sin_port   = __constant_htons(7);
102     my_addr.sin_addr.s_addr = *(unsigned int*)ip;
103
104     if((ret=Angel_Bind(sock, &my_addr, sizeof(my_addr)))<0)
105     {
106         printf("bind error: %d\n", ret);
107         exit(1);
108     }
109     else
110     {
111             printf("bind successful, entering loop\n");
112     }
113
114     // main program loop
115     while(1)
116     {
117         // check if a datagram has arrived
118         if((len=Angel_RecvFrom(sock,buf,BUFLEN,0,&source_addr,&source_len))> 0)
119         {
120             buf[len] = '\0';
121             // print the contents of the datagram and send it back to the source
122             printf("UDP: %s\n", buf);
123             if((ret=Angel_SendTo(sock, buf, len, 0, &source_addr, source_len))<0)
124             {
125                 printf("send error: %d\n", ret);
126             }
127         }
128     }
129
130     return 0;
131 }
132
133
134
135 // send a sample UDP datagram using a given socket, inet_addr is currently
136 // defined in ipstart.c
137 int SendUDPPacket(int sock)
138 {
139     struct sockaddr_in to;
140     char msg[] = "this is a sample udp packet\n";
141
142     /*
143      * open the socket
144      */
145
146     to.sin_family = AF_INET;
147     to.sin_port = __constant_htons(13);
148     to.sin_addr.s_addr = inet_addr(130,237,15,254);     // unicast target
149 //  to.sin_addr.s_addr = inet_addr(255,255,255,255);    // broadcast
150
151     return Angel_SendTo(sock, msg, sizeof(msg), 0, (struct sockaddr*)&to,
sizeof(to));
152 }
```

Listing 35. UDP echo server on the Badge

After inclusion of the necessary header files the OS timer channel on which the software timer interrupt is to run is defined in line 34.

Line 49 sets an IP address in case bootp fails.

After initializing the needed support code modules the IrDA stack is started in line 73 by calling `irda_init()`.

Following that, the function to initialize the IP stack is called. This function is explained in more detail in section 9.6.2.

After initialization and configuration of the IP stack a socket is opened and a function writes out three sample datagrams. Note that the cast used for assigning the address to the "`my_addr`" structure only works in little endian mode, if the program is to be portable or to be used on a big endian system a conversion function has to be called.

Then the socket is bound to the local address and port 7 in line 104.

After successful binding an endless loop is entered which, using the socket function `recvfrom()` constantly checks if a datagram for the specified port has arrived. If so its contents are output to stdout and the datagram is sent back to the source.

This simple example demonstrates all features to build more complex applications. For example it would be straight forward now to write a program that listens on a specific port for certain commands and in response to these streams out sensor values (the code for reading the sensor values could e.g. be taken from the HTTP server running under VxWorks). Using the simple UDP command interface parameters like destination address and port, which sensor values to transmit, update frequency, etc. could be configured dynamically.

In theory it also should be possible to stream audio data from and to the Badge. For this more testing would have to be done regarding the number of available buffers (which at the moment is quite low). Also I found that the IR-link is extremely sensitive to noise which is significantly increased if e.g. a microphone or a loudspeaker is attached to the Badge. But Badge Version 4 is designed to provide a much better shielding between these components and with the memory doubled audio applications via the IR link might get interesting.

9.6.2 The IP Stack Initialization in Detail

```
 1 This section explains the function used to start and configure the IP/UDP stack.
 2 /*
 3  *      Description:
 4  *          Initialize the Angel IP/UDP stack after an IrLAN link has been
 5  *          established by the IrDA stack. After IP stack initialization
 6  *          get an IP address via BOOTP or set a static IP address (depending
 7  *          on Angel configuration
 8  *
 9  *          Note: currently this setup does not support proper restart after
10  *                a link shutdown as the IP stack is only initialized once at
11  *                program start and this also involves setting the filters
12  *                in the access point so after a link shutdown the connection
13  *                will be reestablished but unless ipstack_init is executed
14  *                again the filters will remain disabled.
15  *
16  *      --Christoph Wolf
17  *         chwolf@it.kth.se
18  */
19
20 #include <string.h>
21 #include <stdio.h>
22
23 #include <util/util_misc.h>
24 #include <util/util_debug.h>
```

Listing 36. IP stack initialization code

```
 25 #include <irda/address.h>
 26 #include <irda/netdevice.h>
 27 #include <irda/irlan_common.h>
 28
 29 extern struct device irlan_dev;
 30
 31 extern unsigned char* irlan_recv_balance_buf;
 32 extern unsigned char  irlan_recv_buf[1600];
 33 extern unsigned int   irlan_recv_buf_count;
 34 extern unsigned int   irlan_recv_buf_size;
 35
 36 int irlan_eth_xmit(unsigned char* buff, int size);
 37
 38
 39
 40 int ipstack_init(ip_addr client_ip, ip_addr cur_ip)
 41 {
 42     struct irlan_info_block info;
 43     int ret;
 44     struct irlan_cb *self;
 45     char debug_buf[20];
 46
 47     DEBUG(4, “ipstack_init()\n”);
 48
 49     // wait until unicast address has been received
 50     while(!irlan_dev.nw_stack_init)
 51         ;
 52
 53     // init info block to exchange info between IrDA stack and IP/UDP stack
 54
 55     // transfer mac address from IrDA stack to IP/UDP stack
 56     memcpy(info.hw_address, irlan_dev.dev_addr, MAC_ADDRESS_SIZE);
 57
 58     // set function pointer to be called by irlan_device_write in
 59     // Angel\..\irlan_device.c
 60     info.irlan_eth_xmit = (unsigned int *)irlan_eth_xmit;
 61
 62     // make addresses of IrDA stack receive variables known to IP/UDP
 63     info.irlan_recv_buf  = irlan_recv_buf;
 64     info.irlan_recv_buf_count = &irlan_recv_buf_count;
 65     info.irlan_recv_buf_size = &irlan_recv_buf_size;
 66
 67     // start the IP/UDP stack
 68     if((ret=Angel_NetstartMain(&info))!=0)
 69     {
 70 // fixme
 71         printf(“error in Netstart\n”);
 72         return ret;
 73     }
 74
 75     // get the balance buffer for IrDA - irlan_eth_receive
 76     irlan_recv_balance_buf = info.balance_buf;
 77
 78     DEBUG(2, “Angel_NetstartMain() executed, IP/UDP stack initialized, now setting
filters\n”);
 79
 80     self = (struct irlan_cb *) irlan_dev.priv;
 81     ASSERT(self != NULL, return;);
 82
 83     // both stacks initialized, open unicast filter
 84     irlan_open_unicast_addr(self);
 85
 86     // Open broadcast filter, close multicast filter
 87     irlan_set_broadcast_filter(self, TRUE);
```

Listing 36. IP stack initialization code

```
 88     irlan_set_multicast_filter(self, FALSE);
 89
 90     DEBUG(2, “unicast and broadcast enabled, multicast disabled, ready to go\n”);
 91
 92     // Ready to transfer Ethernet frames
 93     irlan_dev.tbusy = 0;
 94
 95     DEBUG(2, “calling Angel_ConfigureIP()”);
 96     // IP stack is initialized, get/set IP address
 97     if((ret=Angel_ConfigureIP(client_ip))!=0)
 98     {
 99         printf(“error in ConfigureIP\n”);
100         DEBUG(1, “Angel_ConfigureIP(): bootp not successful\n”);
101 //      return ret;
102     }
103
104     Angel_IPGetDeviceAddress(0, cur_ip);
105     sprintf(debug_buf, “%d.%d.%d.%d”, cur_ip[0], cur_ip[1], cur_ip[2], cur_ip[3]);
106     DEBUG_1(2, “Angel_AngelConfigureIP() executed, IP address is %s\n”,
107          debug_buf);
108
109
110     return ret;
111
112     return 0;
113 }
114
115
116 // simple version of inet_addr with changed parameters to convert an IP
117 // given as four bytes to the 32 bit address value
118 unsigned int inet_addr(unsigned char a, unsigned char b,
119                        unsigned char c,unsigned char d)
120 {
121     unsigned char ip[4];
122
123     ip[0] = a;
124     ip[1] = b;
125     ip[2] = c;
126     ip[3] = d;
127     return *(unsigned int*)ip;
128 }
```

Listing 36. IP stack initialization code

Immediately after execution start the function enters a loop checking for a flag in `irlan_dev`. This flag is set in `irlan_client.c` after the loaned unicast address from the access point has been received. At that time the IP stack is ready for initialization.

First the fields of the `irlan_info_block` structure are set. This variable then is passed on to the actual IP stack initialization function (`Angel_NetstartMain()` which is an Angel function made accessible via the standard mechanism. The passed info contains the address of the IrLAN transmit function which is to be called by the low-level write function in the IP stack, and some pointers used to share access to a buffer where the IrLAN receive function deposits newly arrived data and the corresponding IP-stack reading function collects it.

Upon return from the stack initialization the field `info.balance_buf` contains the pointer to a buffer that has been allocated within the IP stack. When receiving data, `irlan_eth_receive()` calls the IP stack function `ethernet_process_one_packet` (accessed through the pointer `Angel_EthernetProcessOnePacket`) to get the packet read and processed. This function expects to be given a buffer and in turn returns an unused buffer, so `irlan_eth_receive()` passes in the pointer to `info.balance_buf` and gets the new buffer back in the same variable, so that it can directly be used again during the next call.

Then the unicast and broadcast filters are opened, while the multicast filters are set to closed. At that time it is safe to request IP configuration via a call to `Angel_ConfigureIP()`, again a function pointer to an internal Angel function. The function expects an IP address to configure the IP stack with, in case bootp is disabled or does not succeed, in which case the supplied IP address will be set.

Finally the current IP address is read by `Angel_IPGetDeviceAddress()` and returned to the user as a pointer in the call to `ipstack_init()`. By reading the return value the caller also can check if a bootp or user supplied IP address was set.

After return from the function the IP stack is ready for use.

A problem is that currently the IP stack is configured only once at program start. While this works fine as long as the IrDA link is established only once this works fine. But if the link is disrupted IrDA will reconnect but as the IP stack initialization is only done at program start, therefore the filters (which are off by default) won't be opened by the access point and no ethernet packets can be sent or received. If the stack has to be capable of dealing with interrupted and resumed connections opening the filters has to be triggered by the IrDA stack upon establishment of a connection. Actually this was the original setup but it turned out that executing the bootp request with its potentially very long timeout (exponential backoff) spent in a busy loop would crash the IrDA stack because the necessary turn around packets couldn't be generated any more. Therefore the code was moved to the main program where it can easily be interrupted. The solution will have to be to trigger the IP stack initialization in the IrDA stack but execute it in its own callback. Then the long timeout shouldn't be a problem.

9.7 Current State, Known Problems, Improvements

The current state of the project is as follows:

The implementation provides a working IrDA stack including IrLAN. Available baudrates range from 9600 to 115200 baud, preparations have been done for adding FIR support. IrLAN is integrated with the Fusion IP/UDP stack which has been changed to interact with the IrDA stack and is compiled into Angel. The IP/UDP stack offers the basic socket functionality, direct access to the UDP functions can be gained by exporting the functions via the mechanism described in section 4.1.1. The IP stack uses BOOTP to request an IP-address at boottime, if BOOTP fails a usersupplied address is configured.

Note: The IrDA stack is a full implementation, containing not only client but also provider functionality. Due to time limits I could not test it but in theory not only access point mode between a Badge and an AccessPoint works but also peer mode between two Badges should work or at least it should be possible to make it run with not too high effort.

Known problems:

At the moment no severe problems are known, basic sending and receiving of UDP datagrams works fine, but I did not have enough time to carry out intensive tests to check behaviour, memory needs and speed under heavy load. There seems to be a little problem with reestablishing the link after a disruption - sometimes it happens that the debug output seems to indicate a connection while the access point does not show a proper link. After disrupting once again normally the connection is properly established.

In Linux the IrDA stack is implemented as a kernel module but controlled by a user space program called irmanager. As this concept is not available on the Badge I tried to integrate the irmanager actions directly into the stack. Possibly the problem with link close-down and reestablishment is located there, the irmanager events are involved in shutting down a connection.

A second issue is that at the moment the IP stack is initialized by the main program after the IrDA stack has been set up and set a flag indicating a connection. So after link shut down the IP stack currently is not reinitialized.

Possible improvements, further work:

The solution works, but is far from being optimal. In the given amount of time I managed to provide a working platform but the code could be heavily optimized. The code has been ported to the Badge under Angel but the Linux structure has been widely kept, as the stress was on getting a working platform first, so the code should be changed as few as possible. Now starting with this working implementation it could be refined step by step by removing further code that is not needed on the Badge. In this context a minimal version could be built (without provider functionality) to minimize the memory needs. Also some functions could be integrated into one the other to avoid having too many layers of function-calls which costs performance on this type of processor as it potentially breaks the pipeline.

Another issue is the interface between the IrDA stack and the IP stack. As described before the memory management is completely different and thus an inefficient interface results. Better integration of the two stacks might improve its performance.

10. Conclusions and Further Work

In this chapter I will present my conclusions and give a few ideas for further work.

After an introduction to the underlying hardware platform and the available operating systems I examined a few of the units provided by the StrongARM microprocessor in more detail. The work in these topics resulted in a small library that allows easy use of the most important units such as serial I/O, general purpose I/O, interrupts, timers. It also contains functions and declarations to aid in the debugging process and to access internal Angel functions which are not directly accessible otherwise. This was an important base for the further work that relied heavily on the possibility to queue time consuming tasks in interrupt handlers for later execution.

I further did some tests concerning the code generation, the influence of the caches and upper bounds on the toggling frequency for I/O-pins and discussed various methods of debugging embedded systems. One chapter deals with the possibility to do infrared transmission using software modulation instead of the erroneous hardware modulation unit in the original StrongARM processor. Using the results from the earlier work concerning the caches I could show that basically it is possible to do modulation under software control at 9600 baud. It turned out though that under heavy interrupt load as in the case of the IrDA protocols software modulated transmission was not reliable.

The main part of my work was to implement the IrDA protocol stack on the Badge. The result is a port of the Linux IrDA kernel module to the Badge. I have integrated the IrDA stack with the Fusion IP/UDP stack which now allows to acquire an IP address at boottime using the BOOTP protocol and following the setup to send and receive UDP packets using the socket interface. The chapter describing my implementation also describes which parts have to be looked at in particular if the stack has to be ported to another operating system on the Badge.

In discussions with my advisor Prof. Gerald Maguire and during my work various ideas for additional tasks turned up. Due to lack of time some of them could not be done in the scope of this work. I will present a few of them in the following lines:

- Section 4.5.3, “Maximum Toggling Frequency of GPIO and PPC Pins,” on page 50:

  It would be interesting to investigate if the difference in execution time between the standalone SRAM and the Angel SRAM version actually is related to the MMU. This could be examined by building a standalone version that enables the MMU.

- In section 4.6.2 the trimming of the RTC clock was discussed. While this involves a manual trimming one could imagine a setup to allow automatic trimming of the RTC clock. This could be done by feeding a know frequency at one of the GPIO pins and on another pin feeding back the signal output at GPIO27 which is the internal clock rate. Using interrupt driven edge detection one could count the transitions and by comparing them could automatically compute the trim factor. By extending the measurement time the precision can easily be made as high as necessary to have full precision as offered by the limited precision trim factor.
- In Section 4.8.4.6 I talked about detecting the end of a transmission: The PPC contains a register that controls and monitors the pin state (PPSR). This register can be read at any time, even if the pin is under control of a peripheral. Thus it should be possible to monitor the activity on the transmit pin to detect the end of the transmission instead of waiting in a fixed loop.
- Angel supports debugging via Ethernet. Now with the IrDA stack running this could be used to do debugging via FIR as soon as FIR is supported. Apart from the FIR driver this would only require one more file (arm_ether.c which maps the between the network stack and the Angel device driver framework). This would allow to do the debugging, especially downloading of files, at a much higher rate than is currently possible using the serial link.
- It would be interesting to carry out measurements on the CPU load overhead that the IrDA protocols impose. This could be done by running in a loop and counting how often the loop is executed in a certain time (measured using the OS timer count register). By comparing that to a setup without running the IrDA protocols one can deduct a lower bound caused by the continuos receiver-ready messages that have to be exchanged. By then starting to send network packets one can measure the overhead caused by passing through all the layers and statemachines.

- I have not had the time to calculate the memory requirements - the linker map could be used to calculate the size for the different targets and the different IrDA layers.
- In the case of IrLAN transporting only IP traffic the question arose if IrTTP really is necessary. Only one channel is use, therefore deadlocks shouldn't be a problem. As far as packet loss is concerned, if a UDP application needs reliable transmission it has to implement its own protocol anyway. And the case of segmentation can be solved by setting the MTU of the IP stack. So it might be possible to remove IrTTP and just fake the packets in order to have the access point, which of course requires IrTTP compliant packets accept them. But locally a complete layer could be saved which should increase the performance.
- More testing should be done with the IrDA implementation and as already mentioned by refining the base code it should be possible to shrink the size. Also it should be tested how the stack behaves under heavy load and how its behaviour is influenced by the amount of buffers (which are currently set to a very small number in order to save memory). Moreover I just took the link configuration values (IrLAP) as set in Linux. One could experiment if changing the turn-around-times, window size, etc. can improve the performance.

References

[1] Mark T. Smith, "SmartCards: Integrating for Portable Complexity", IEEE Computer, August 1998, pp. 110-112, and 115.

[2] PHILIPS, “UCB1200 Advanced modem/audio analog front-end DATA SHEET”, August 1997

[3] Advanced RISC Machines Ltd (ARM), “ARM Software Development Toolkit v.211, Reference Guide”, Document Number: ARM DUI 0041B, June 1997

[4] Advanced RISC Machines Ltd (ARM), “ARM Software Development Toolkit v.211, User Guide”, Document Number: ARM DUI 0040C, May 1997

[5] Dave Jaggar, “ARM Architectural Reference Manual”, Prentice Hall, July 1996

[6] Intel Corporation, “Intel StrongARM SA-1100 Microcprocessor Developer’s Manual”, August 1999

[7] Intel Corporation, “Intel StrongARM SA-1110 Microprocessor, Advanced Developer’s Manual”, December 1999

[8] Intel Corporation, “Intel StrongARM SA-1111 Microprocessor Companion Chip, Developer’s Manual”, August 1999

[9] Malena Mesarina, “How to port Angel to Badgepad”, Hewlett-Packard Research Laboratories, Palo Alto, California, first released September 1998, last revised March 2000

[10] T. Simunic, L. Benini, and G. De Micheli, "Dynamic Power Management of Portable Systems", to appear at Mobicom 2000.

[11] Infrared Data Association, “Serial Infrared Physical Layer Specification”, Version 1.3, October 1998

[12] Infrared Data Association, “Serial Infrared Link Access Protocol (IrLAP)”, Version 1.1, June 1996

[13] Infrared Data Association, “Link Management Protocol”, Version 1.1, Jannuary 1996

[14] Infrared Data Association, “‘Tiny TP’: A Flow-Control Mechanism for use with IrLMP”, Version 1.1, October 1996

[15] Infrared Data Association, “LAN Access Extensions for Link Management Protocol IrLAN”, Version 1.0, July 1997

[16] http://www.irda.org/standards/standards.asp

Acronyms and Abbreviations

4PPM	Four-position pulse modulation
ADC	Analog-Digital-Converter
APCS	ARM Procedure Call Standard
CODEC	Coder/decoder
DAA	Data access arrangement
DMA	Direct Memory Access
DRAM	Dynamic random access memory
FIFO	First-In-First-Out
FIR	Fast Infrared Mode
FLASH-RAM	Electrically alterable read-only memory
FSM	Finite State Machine
GPIO	General Purpose Input/Output
HDLC	High-Level Data Link Control
IAP	Information Access Protocol
IAS	Information Access Service
IP	Internet Protocol
IC	Instruction Cache
IrDA	Infrared Data Association
IrLAN	Infrared LAN
IrLAP	Infrared Link Access Protocol
IrLMP	Infrared Link Management Protocol
IrPHY	Infrared Physical Layer
IrTTP	Infrared Tiny Transport Protocol
ISAP	IrLAP Service Access Point
LM-MUX	Link Management Multiplexer
LSAP	Link Service Access Point
LSAP-Sel	Link Service Access Point Selector
LSB	Least significant bit/byte
MAC	Media Access Control
MMU	Memory Management Unit
MSB	Most significant bit/byte
PCMCIA	Personal computer memory card interface adapter
PDU	Protocol Data Unit
PPC	Peripheral Pin Controller
RISC	Reduced Instruction Set Computer
RTC	Real Time Clock
SDLC	Synchronous Data Link Control
SDRAM	Synchronous DRAM
SDU	Service Data Unit

SIR	Hewlett-Packard Serial Infrared Standard
SRAM	Static random access memory
TSAP	IrTTP Service Access Point
TTPSAP	IrTTP Service Access Point
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
USB	Universal Serial Bus
XID	Exchange Station Identification

Appendix A. Infrared support code

This chapter contains the listings for the source code related to infrared communication as described in Chapter 5. ”Infrared Communication”. The code is split into three files, a common header file (util_ir.h) and two C-files, containing the functions to support SIR-mode (both software modulation and UART based, util_sir.c) and FIR-mode (util_fir.c).

A.1 The Header File, Containing Constants and Macros

To keep the listing short only the pure function prototypes are printed here while the corresponding descriptions have been removed. They are contained in the header file `util_ir.h` though and they are included in the listings of the two C-files in the following sections.

```
/*
 *      Description:
 *          Type declarations and defines for use with the infrared code.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#ifndef util_ir_h
#define util_ir_h

#include <util/util_misc.h>
#include <util/util_gpio.h>
#include <util/util_ppc.h>
#include <util/util_ringbuf.h>

#ifndef SIR_UART_TRANSMISSION
    #include <util/util_ostimer.h>
#endif



#define IR_UART_BASE UART2_BASE    /* UART (Infrared port low speed) base */

#define HSSP_BASE   0x80040060     /* HSSP   (Infrared port high speed) base */

/* HSSP (High speed infrared) register offsets from base address */
#define HSCR0           0x00    /* HSSP Control Register 0 */
#define HSCR1           0x04    /* HSSP Control Register 1 */
#define HSDR            0x0C    /* HSSP Data Register */
#define HSSR0           0x14    /* HSSP Status Register 0 */
#define HSSR1           0x18    /* HSSP Status Register 1 */

#define HSCR2           0x28    /* HSSP Control Register 2 (located */
                                /* in PPC address space !!!!!!!!!   */
                                /* complete address 0x9006 0028)    */

/* HSSP control register 0 bits */
#define HSCR0_ITR   1   /* IrDA transmission rate */
#define HSCR0_LBM   2   /* Loopback Mode */
#define HSCR0_TUS   4   /* Transmit FIFO Underrun Select */
#define HSCR0_TXE   8   /* Transmit Enable*/
#define HSCR0_RXE   16  /* Receive Enable */
#define HSCR0_RIE   32  /* Receive FIFO Interrupt Enable */
```

Listing 37. Constants, macros and function prototypes for IR on the SmartBadge

```
#define HSCR0_TIE   64  /* Transmit FIFO Interrupt Enable */
#define HSCR0_AME   128 /* Address Match Enable */

/* HSSP control register 2 bits */
#define HSCR2_TXP   4   /* Transmit Pin Polarity Select */
#define HSCR2_RXP   8   /* Receive Pin Polarity Select */

/* HSSP status register 0 bits */
#define HSSR0_EIF   1   /* Error in FIFO */
#define HSSR0_TUR   2   /* Transmit FIFO Underrun*/
#define HSSR0_RAB   4   /* Receiver Abort */
#define HSSR0_TFS   8   /* Transmit FIFO Service Request */
#define HSSR0_RFS   16  /* Receive FIFO Service Request */
#define HSSR0_FRE   32  /* Framing Error */

/* HSSP status register 1 bits */
#define HSSR1_RSY   1   /* Receiver Synchronized Flag */
#define HSSR1_TBY   2   /* Transmitter Busy Flag*/
#define HSSR1_RNE   4   /* Receive FIFO not empty  */
#define HSSR1_TNF   8   /* Transmit FIFO not full */
#define HSSR1_EOF   16  /* End of Frame  */
#define HSSR1_CRE   32  /* CRC Error */
#define HSSR1_ROR   64  /* Receive FIFO Overrun */


/* GPIO 26 used as mode select for IrDA transceiver */
#define IRSD    GPIO26



/*****************************************************************************/
/*                     macros related to serial port 2                       */
/*****************************************************************************/

/* select HP-SIR modulation (UART) or 4PPM modulation (HSSP) / including
 * setting of the mode pin for the transceiver.
 */
#define IR_SEL_SIR        REG(HSSP_BASE,HSCR0) &= ~0x1;   \
                                GPIO_SET_OUTPUT(IRSD);GPIO_CLEAR_PIN(IRSD)

#define IR_SEL_FIR        REG(HSSP_BASE,HSCR0) |= 0x1;    \
                                GPIO_SET_OUTPUT(IRSD);GPIO_SET_PIN(IRSD)


/* set the polarity of the TxD2, RxD2 pins to true (non-inverted) or inverted */
/* Note: HSSP and UART must be disabled (TXE=RXE=0) when changing the state
 *       of these bits !!!
 * Note: FIR mode on the Badge only seems to work if RXD2 is inverted !!
 */
#define IR_SET_POLARITY_TXD2_TRUE       (REG(PPC_BASE, HSCR2) |=  BIT18)
#define IR_SET_POLARITY_TXD2_INVERTED   (REG(PPC_BASE, HSCR2) &= ~BIT18)
#define IR_SET_POLARITY_RXD2_TRUE       (REG(PPC_BASE, HSCR2) |=  BIT19)
#define IR_SET_POLARITY_RXD2_INVERTED   (REG(PPC_BASE, HSCR2) &= ~BIT19)



/*****************************************************************************/
/*                macros related to serial port 2 UART mode                  */
/*****************************************************************************/

/* SER_UART_IRDA_SIR_*ABLE always enables low power mode, and dis/enables
   the SIR mode of serial port 2 (SIR or normal UART modulation) */

// !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
// before changing the bits in UTCR4, transmitter and receiver !
```

Listing 37. Constants, macros and function prototypes for IR on the SmartBadge

```
// have to be disabled                                              !
// !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
#define IR_UART_SIR_ENABLE    (REG(UART2_BASE,UTCR4) = 0x3)
#define IR_UART_SIR_DISABLE   (REG(UART2_BASE,UTCR4) = 0x2)



/*****************************************************************************/
/*                macros related to serial port 2 HSSP mode                  */
/*****************************************************************************/

/* (disable UART 2), en/disable HSSP transmitter and/or receiver
 * (use 1/0 as arguments)
 */
#define IR_HSSP_EN(txd, rxd)  \
    (REG(HSSP_BASE,HSCR0) = 1 | (txd)<<3 | (rxd)<<4)

/* completely shutdown the HSSP and give TxD2-,RxD2-control to the PPC if
 * UART2 is not enabled.
 */
#define IR_HSSP_SHUTDOWN (REG(HSSP_BASE, HSCR0) = 0x0)

/* set the address match value in control register 1 */
#define IR_HSSP_SET_ADDRESS(addr) (REG(HSSP_BASE,HSCR1) = addr)

/* en-/disable HSSP loopback */
#define IR_HSSP_ENABLE_LOOPBACK (REG(HSSP_BASE,HSCR0) |= HSCR0_LBM)
#define IR_HSSP_DISABLE_LOOPBACK (REG(HSSP_BASE,HSCR0) &= ~HSCR0_LBM)

/* set transmitter underrun behaviour (end frame regularly/abort frame) */
#define IR_HSSP_TUS_ABORT  (REG(HSSP_BASE,HSCR0) |= HSCR0_TUS)
#define IR_HSSP_TUS_END    (REG(HSSP_BASE,HSCR0) &= ~HSCR0_TUS)

/* enable receiver and/or transmitter  */
#define IR_HSSP_ENABLE_R (REG(HSSP_BASE,HSCR0) |= HSCR0_RXE)
#define IR_HSSP_ENABLE_T (REG(HSSP_BASE,HSCR0) |= HSCR0_TXE)
#define IR_HSSP_ENABLE_TR (REG(HSSP_BASE,HSCR0) |= (HSCR0_RXE | HSCR0_TXE))

/* disable HSSP receiver and/or transmitter */
#define IR_HSSP_DISABLE_R REG(HSSP_BASE,HSCR0) &= ~HSCR0_RXE
#define IR_HSSP_DISABLE_T REG(HSSP_BASE,HSCR0) &= ~HSCR0_TXE
#define IR_HSSP_DISABLE_TR REG(HSSP_BASE,HSCR0) &= ~(HSCR0_RXE | HSCR0_TXE)

/* enable HSSP receiver and/or transmitter interrupt */
#define IR_HSSP_ENABLE_RI (REG(HSSP_BASE,HSCR0) |= HSCR0_RIE)
#define IR_HSSP_ENABLE_TI (REG(HSSP_BASE,HSCR0) |= HSCR0_TIE)
#define IR_HSSP_ENABLE_TRI (REG(HSSP_BASE,HSCR0) |= (HSCR0_RIE | HSCR0_TIE))

/* disable HSSP receiver and/or transmitter interrupt */
#define IR_HSSP_DISABLE_RI (REG(HSSP_BASE,HSCR0) &= ~HSCR0_RIE)
#define IR_HSSP_DISABLE_TI (REG(HSSP_BASE,HSCR0) &= ~HSCR0_TIE)
#define IR_HSSP_DISABLE_TRI (REG(HSSP_BASE,HSCR0) &= ~(HSCR0_RIE | HSCR0_TIE))

/* en-/disable HSSP address match functionality */
#define IR_HSSP_ENABLE_ADDR_MATCH (REG(HSSP_BASE,HSCR0) |= HSCR0_AME)
#define IR_HSSP_DISABLE_ADDR_MATCH (REG(HSSP_BASE,HSCR0) &= ~HSCR0_AME)

#define IR_HSSP_SET_ADDR_MATCH(amv) (REG(HSSP_BASE, HSCR1) = (amv))
#define IR_HSSP_GET_ADDR_MATCH      (REG(HSSP_BASE, HSCR1))

/* macros to clear sticky status bits in status register 0 */
#define IR_HSSP_CLEAR_STATUS_BITS_ALL   (REG(HSSP_BASE, HSSR0) = 0xFF)
#define IR_HSSP_CLEAR_STATUS_BITS_TUR   (REG(HSSP_BASE, HSSR0) = BIT1)
#define IR_HSSP_CLEAR_STATUS_BITS_RAB   (REG(HSSP_BASE, HSSR0) = BIT2)
#define IR_HSSP_CLEAR_STATUS_BITS_FRE   (REG(HSSP_BASE, HSSR0) = BIT5)
```

Listing 37. Constants, macros and function prototypes for IR on the SmartBadge

```
/* macros for querying the status flags in status register 1 */
#define IR_HSSP_REC_SYNC_Q (REG(HSSP_BASE,HSSR1) & HSSR1_RSY)
#define IR_HSSP_TRANSM_BUSY_Q (REG(HSSP_BASE,HSSR1) & HSSR1_TBY)
#define IR_HSSP_REC_NOT_EMPTY_Q (REG(HSSP_BASE,HSSR1) & HSSR1_RNE)
#define IR_HSSP_TRANSM_NOT_FULL_Q (REG(HSSP_BASE,HSSR1) & HSSR1_TNF)
#define IR_HSSP_EOF_Q (REG(HSSP_BASE,HSSR1) & HSSR1_EOF)
#define IR_HSSP_CRC_ERROR_Q (REG(HSSP_BASE,HSSR1) & HSSR1_CRE)
#define IR_HSSP_REC_OVERRUN_Q (REG(HSSP_BASE,HSSR1) & HSSR1_ROR)

/* read/write a byte directly from/to HSSP data register */
#define IR_HSSP_PUT_BYTE_DIRECT(ch)   (REG(HSSP_BASE,HSDR) = ch)
#define IR_HSSP_GET_BYTE_DIRECT       (REG(HSSP_BASE,HSDR))


/*****************************************************************************/
/*                           FIR function prototypes                         */
/*****************************************************************************/

void fir_Init(BOOL receive_int_en, BOOL transm_int_en, BOOL addr_match_en,
         UC8 addr_match_val);


/* constants to be used as parameters for fir_Init */
#define HSSP_INIT_RECEIVE_INT_DIS  0
#define HSSP_INIT_RECEIVE_INT_EN   1

#define HSSP_INIT_TRANSMIT_INT_DIS 0
#define HSSP_INIT_TRANSMIT_INT_EN  1

#define HSSP_INIT_ADDR_MATCH_DIS   0
#define HSSP_INIT_ADDR_MATCH_EN    1

void fir_PutBytePolled(unsigned char ch);
unsigned char fir_GetBytePolled(void);


/*****************************************************************************/
/*                 general SIR declarations and functions                    */
/*****************************************************************************/

void sir_Init(int );
void sir_SetReceiveMode(void);
void sir_SetTransmitMode(void);
int sir_EnqueueTransmData(UC8* data, UC8 length);
void sir_EnqueueTransmByte(UC8 byte);
int ir_SirQueueDataMultiple(UC8 val, int count);
BOOL sir_NewReceiveData_Q(void);
int sir_GetReceiveCount(void);
int sir_GetReceiveData(UC8* data, int len);


/*****************************************************************************/
/*                     UART SIR declarations and functions                   */
/*****************************************************************************/

/*
 * delay to allow transmitter busy flag to be set after start of
 * transmission.
 */
#define SIR_TRANSM_BUSY_DELAY 200

void sir_RestartTransmUART(void);
```

Listing 37. Constants, macros and function prototypes for IR on the SmartBadge

```
/*****************************************************************************/
/*      declarations and functions related to SIR software modulation        */
/*****************************************************************************/

extern RingBuffer sir_transmit_buf;
extern RingBuffer sir_receive_buf;
extern RingBuffer sir_sw_debug_buf;

#define IR_OSM_FACTOR   24
#define SIR_OSTIMER_CHANNEL OSTIMER_CHANNEL_3

#define RX_BUF_SIZE 12

#define FIRST_PERIOD    0
#define SECOND_PERIOD   1

#define SIR_TRANSMIT_BUF_SIZE    2000
#define SIR_RECEIVE_BUF_SIZE    2000
#define SIR_SW_DEBUG_BUF_SIZE       2000


void sir_TransmitDataSW(void);
BOOL sir_SWTransmitting_Q(void);
void sir_PrintDebugBuf(void);

#endif
```

Listing 37. Constants, macros and function prototypes for IR on the SmartBadge

A.2 Implementation of the SIR-Functions

```
/*
 *      Description:
 *          Code to support infrared communication in SIR mode on the Badge.
 *          SIR operation can be changed between software modulation and
 *          use of the serial port 2 UART on newer revisions of the SA-1100
 *          by commenting/uncommenting the symbol SIR_UART_TRANSMISSION. When
 *          code in this file is used it has to be made sure that util_sir.c
 *          is compiled with the right setting for the intended use. Otherwise
 *          the results are unpredictable !
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */


/*
 * configuration (do software or hardware transmission, enable software
 * modulation counters, enable debug output into a ringbuffer during
 * software modulation).
 * An option is enabled if the symbol is defined and disabled if the
 * symbol definition is commented out.
 */
#define SIR_UART_TRANSMISSION
//#define SIR_DEBUG_SOFTWARE_SIR
//#define SIR_SW_DEBUG_BUF


#include <stdio.h>
```

Listing 38. Implementation of SIR-related functions

```
#include <util/util_ir.h>
#include <util/util_ppc.h>
#include <util/util_serial.h>
#include <util/util_interrupt.h>
#include <util/util_misc.h>
#include <util/util_ostimer.h>
#include <util/util_ringbuf.h>
#include <util/util_debug.h>

/* external variables to check if certain modules have been initialized */
extern int misc_initialized;
extern int debug_initialized;

/* The SIR interrupt handler */
void sir_IntHandler(unsigned int ident, unsigned int data,
                      unsigned int empty_stack);



/* Switch on or off debug output */
#ifdef SIR_DEBUG_SOFTWARE_SIR
    #define DEBUG_STRING(c) debug_String(c)
    #define DEBUG_PUT_BYTE(c) debug_PutByte(c)
    #define DEBUG_PUT_BYTE_DIRECT(c) SER_UART_PUT_BYTE_DIRECT(DEBUG_UART_BASE, c)
    #define DEBUG_PUT_BYTE_HEX(c) debug_PutByteHex(c)
#else
    #define DEBUG_STRING(c)
    #define DEBUG_PUT_BYTE(c)
    #define DEBUG_PUT_BYTE_DIRECT(c)
    #define DEBUG_PUT_BYTE_HEX(c)
#endif


/* Buffer to contain debug output from the software modulation function
   to allow an exact trace of the executed sequence.
 */
#ifdef SIR_SW_DEBUG_BUF
    RingBuffer sir_sw_debug_buf;
#endif


/* The SIR transmission and receive buffers */
RingBuffer sir_transmit_buf;
RingBuffer sir_receive_buf;
int sir_transmit_buf_size = SIR_TRANSMIT_BUF_SIZE;
int sir_receive_buf_size = SIR_RECEIVE_BUF_SIZE;

BOOL sir_new_receive_data;

/* Debug counter variables for software modulation */
#ifdef SIR_DEBUG_SOFTWARE_SIR
    int sir_int_total = 0;
    int sir_int_restart = 0;
    int sir_int_3=0;
    int sir_int_16=0;
    int sir_int_13=0;
    int sir_int_mask=0;
#endif



/*
 * Function: sir_Init
 *  Purpose: Initialize serial port 2 for SIR mode
 *
```

Listing 38. Implementation of SIR-related functions

```
 * Parameters:
 *      Input: baudrate         the desired baudrate, if in UART mode
 *             ostimer_channel  the timer channel to be used if in software
 *                              modulation mode
 *
 *    Returns: void
 *
 * The behaviour of this function is determined by the symbol SIR_UART_TRANSMISSION.
 * If it is defined, the function initializes serial port 2 UART to use the
 * given baudrate and does all other necessary setup.
 * If the symbol is not defined the function sets up everything for software
 * modulation.
 */
#ifdef SIR_UART_TRANSMISSION    // use UART for SIR transmission
    void sir_Init(int baudrate)

#else

    // default initialization of global software modulation variables
    int sir_ostimer_channel =  SIR_OSTIMER_CHANNEL; // timer channel to use
    int sir_ostimer_int = INT_OSTIMER_0 + SIR_OSTIMER_CHANNEL;
    volatile BOOL sir_sw_transmitting = FALSE;

    // the interrupt handler for software modulation transmission.
    void sir_SWTransmitIntHandler(unsigned int ident, unsigned int data,
                                  unsigned int empty_stack);

    void sir_Init(int ostimer_channel)  // use software modulation for transmission
#endif
    {
        // Allocate memory for the transmit and receive buffers and for the
        // software modulation debug buffer, if debug is enabled.
        sir_transmit_buf.data = (unsigned char*) malloc(sir_transmit_buf_size);
        if(sir_transmit_buf.data)
        {
            sir_transmit_buf.size = sir_transmit_buf_size;
        }
        else
        {
            printf("not enough memory for ir transmit buffer\n");
            exit(1);
        }

        sir_receive_buf.data = (unsigned char*) malloc(sir_receive_buf_size);
        if(sir_receive_buf.data)
        {
            sir_receive_buf.size = sir_receive_buf_size;
        }
        else
        {
            printf("not enough memory for ir receive buffer\n");
            exit(1);
        }

    #ifdef SIR_SW_DEBUG_BUF
        sir_sw_debug_buf.data = (unsigned char*) malloc(SIR_SW_DEBUG_BUF_SIZE);
        if(sir_sw_debug_buf.data)
        {
            sir_sw_debug_buf.size = SIR_SW_DEBUG_BUF_SIZE;
        }
        else
        {
            printf("not enough memory for debug buffer\n");
            exit(1);
        }
```

Listing 38. Implementation of SIR-related functions

```
    #endif

        // to make sure, shutdown HSSP unit
        REG(HSSP_BASE, HSCR0) = 0;

        // just to make sure, in case FIR mode has been used before
        IR_SET_POLARITY_RXD2_TRUE;

        // set SIR mode for serial port 2 and set the mode pin accordingly
        IR_SEL_SIR;
        IR_UART_SIR_ENABLE;

    #ifdef SIR_UART_TRANSMISSION

        // setup the UART for the given baudrate and the required frame
        // format
        ser_UartInit(IR_UART_BASE, baudrate, SER_INIT_PARITY_DIS,
                       SER_INIT_PARITY_ODD, SER_INIT_ONE_STOP_BIT,
                       SER_INIT_DATA_SIZE_8, SER_INIT_RECEIVE_INT_EN,
                       SER_INIT_TRANSMIT_INT_DIS);

    #else
        // install software modulation code
        PPC_SET_OUTPUT(TXD2);
        sir_ostimer_channel = ostimer_channel;
        OSTIMER_ENABLE_INT(sir_ostimer_channel);
        sir_ostimer_int = INT_OSTIMER_0+sir_ostimer_channel;
        int_InstallHandler(sir_ostimer_int, sir_SWTransmitIntHandler);

        if(!misc_initialized)
        {
            misc_InitAngelFunctions();
        }

        // seems to be necessary for proper function
        misc_SysDisableWriteBuffer();

        // setup the UART for 9600 baud receive operation
        ser_UartInit(IR_UART_BASE, SER_INIT_BAUD_9600, SER_INIT_PARITY_DIS,
                       SER_INIT_PARITY_ODD, SER_INIT_ONE_STOP_BIT,
                       SER_INIT_DATA_SIZE_8, SER_INIT_RECEIVE_INT_EN,
                       SER_INIT_TRANSMIT_INT_DIS);

    #endif
        // install the interrupt handler for the UART
        int_InstallHandler(INT_UART2, sir_IntHandler);

        if(!debug_initialized)
        {
            debug_Init();
        }

        return;
    }


// the code for software modulation
#ifndef SIR_UART_TRANSMISSION

    volatile static unsigned char sir_sw_current_byte;
    volatile static unsigned int sir_sw_state = FIRST_PERIOD;
    static int sir_sw_bit_count=10;

   /*
    * Function: sir_SWTransmitIntHandler
```

Listing 38. Implementation of SIR-related functions

```
    *  Purpose: Do software modulation to provide SIR transmission
    *
    * Parameters:
    *      Input: interrupt handler parameters suppplied by Angel
    *
    *    Returns: void
    *
    * This is an interrupt handler for one of the ostimer channels. When the timer
    * match first occurs after unmasking the interrupt the handler tries to read a
    * byte out of the SIR transmission ring buffer. This byte then is transmitted
    * by modulating the TXD2 pin under PPC control according to the SIR standard.
    * This is continued until the transmit buffer is empty, in which case the
    * interrupt is masked.
    */
    void sir_SWTransmitIntHandler(unsigned int ident, unsigned int data, unsigned int
empty_stack)
    {
        #ifdef SIR_DEBUG_SOFTWARE_SIR
            sir_int_total++;
        #endif

        /* last byte finished, get new byte */
        if(sir_sw_bit_count>=10 && ringbuf_GetCountInt(&sir_transmit_buf)>0)
        {
            sir_sw_current_byte = ringbuf_ReadByte(&sir_transmit_buf);
            DEBUG_PUT_BYTE_DIRECT(sir_sw_current_byte);

            sir_sw_bit_count=0;
            sir_sw_state = FIRST_PERIOD;

            #ifdef SIR_DEBUG_SOFTWARE_SIR
                sir_int_restart++;
            #endif

            #ifdef SIR_SW_DEBUG_BUF
                ringbuf_WriteByte(&sir_sw_debug_buf, 'r');
            #endif
        }

        if(sir_sw_state==FIRST_PERIOD)
        {
            // start bit or zero data bit -> generate 3/16th pulse
            if(sir_sw_bit_count==0 ||
                (!(sir_sw_current_byte & 1<<(sir_sw_bit_count-1)) && sir_sw_bit_count
<= 8)   )
            {
                OSTIMER_INC_MATCH_REG(sir_ostimer_channel, IR_OSM_FACTOR*3);

                // set pin high
                PPC_SET_PIN(TXD2);

                sir_sw_state = SECOND_PERIOD;

                #ifdef SIR_DEBUG_SOFTWARE_SIR
                    sir_int_3++;
                #endif
                #ifdef SIR_SW_DEBUG_BUF
                    ringbuf_WriteByte(&sir_sw_debug_buf, 'H');
                #endif
            }
            else    // data bit is 1 -> zero period or it is the stop bit
            {
                if(sir_sw_bit_count == 9)
                {   // longer pause after last bit
```

Listing 38. Implementation of SIR-related functions

```
                    OSTIMER_INC_MATCH_REG(sir_ostimer_channel, IR_OSM_FACTOR*48);
                }
                else
                {
                    OSTIMER_INC_MATCH_REG(sir_ostimer_channel, IR_OSM_FACTOR*16);
                }

                sir_sw_bit_count++;

                #ifdef SIR_DEBUG_SOFTWARE_SIR
                    sir_int_16++;
                #endif
                #ifdef SIR_SW_DEBUG_BUF
                    ringbuf_WriteByte(&sir_sw_debug_buf, '0');
                #endif
            }
        }
        else    // state SECOND_PERIOD
        {
            // 13/16th low time
            OSTIMER_INC_MATCH_REG(sir_ostimer_channel, IR_OSM_FACTOR*13);

            // set the pin low
            PPC_CLEAR_PIN(TXD2);
            sir_sw_state = FIRST_PERIOD;
            sir_sw_bit_count++;

            #ifdef SIR_DEBUG_SOFTWARE_SIR
                sir_int_13++;
            #endif
            #ifdef SIR_SW_DEBUG_BUF
                ringbuf_WriteByte(&sir_sw_debug_buf, 'L');
            #endif
        }
        // clear the interrupt request flag
        OSTIMER_RESET_INT(sir_ostimer_channel);

        // byte finished and transfer of buffer finished, disable interrupt */
        if(sir_sw_bit_count>=10 && ringbuf_GetCountInt(&sir_transmit_buf)==0)
        {
            INT_MASK(sir_ostimer_int);
            sir_sw_transmitting=FALSE;

            #ifdef SIR_DEBUG_SOFTWARE_SIR
                sir_int_mask++;
            #endif
            #ifdef SIR_SW_DEBUG_BUF
                ringbuf_WriteByte(&sir_sw_debug_buf, 'm');
            #endif
        }
    }


/*
 * Function: sir_SWTransmitting_Q
 *  Purpose: check if the software modulation based transmission is finished
 *
 * Parameters:
 *      Input: none
 *
 *    Returns: A boolean value telling, if the transmission is still going on
 *             or has been finished.
 */
BOOL sir_SWTransmitting_Q(void)
{
```

Listing 38. Implementation of SIR-related functions

```
    return sir_sw_transmitting;
}

#endif /* SIR_UART_TRANSMISSION */



/*
 * Function: sir_IntHandler
 *  Purpose: The general SIR interrupt handler.
 *
 * Parameters:
 *      Input: Interrupt handler arguments supplied by Angel.
 *
 *    Returns: void
 *
 * This handler stores received bytes into the receive ringbuffer and
 * takes bytes out of the transmit ringbuffer to transmit them. When
 * the transmit buffer is empty the transmit engine is shutdown.
 * Note: the code related to transmission is only compiled if the
 *       symbol SIR_UART_TRANSMISSION is defined, otherwise no transmit
 *       interrupt should occur. If it does, the interrupt will be shutdown.
 */
void sir_IntHandler(unsigned int ident, unsigned int data, unsigned int empty_stack)
{
    if(TEST_BIT(IR_UART_BASE, UTSR0, UTSR0_RFS))  // receiver fifo request
    {
        ringbuf_WriteByte(&sir_receive_buf, REG(IR_UART_BASE, UTDR));
        ringbuf_WriteByte(&sir_receive_buf, REG(IR_UART_BASE, UTDR));
        ringbuf_WriteByte(&sir_receive_buf, REG(IR_UART_BASE, UTDR));
        ringbuf_WriteByte(&sir_receive_buf, REG(IR_UART_BASE, UTDR));
        DEBUG_PUT_BYTE('r');
        sir_new_receive_data = TRUE;
    }

    else if(TEST_BIT(IR_UART_BASE, UTSR0, UTSR0_RID)) // receiver idle
    {
        do
        {
            ringbuf_WriteByte(&sir_receive_buf, REG(IR_UART_BASE, UTDR));
            DEBUG_PUT_BYTE('i');
        }
        while(SER_UART_REC_NOT_EMPTY_Q(IR_UART_BASE));
        SER_UART_CLEAR_STATUS_BITS_RID(IR_UART_BASE);

        sir_new_receive_data = TRUE;
    }

    else if(TEST_BIT(IR_UART_BASE, UTSR0, UTSR0_RBB)) // receiver break begin
    {
        SER_UART_CLEAR_STATUS_BITS_RBB(IR_UART_BASE);
    }
    else if(TEST_BIT(IR_UART_BASE, UTSR0, UTSR0_REB)) // receiver break end
    {
        SER_UART_CLEAR_STATUS_BITS_REB(IR_UART_BASE);
    }

    else
#ifdef SIR_UART_TRANSMISSION
    if(TEST_BIT(IR_UART_BASE, UTSR0, UTSR0_TFS))  // transmit fifo request
    {
        int i, count;

        DEBUG_PUT_BYTE('t');
```

Listing 38. Implementation of SIR-related functions

```
        // get number of characters left in the transmit buffer
        count = ringbuf_GetCountInt(&sir_transmit_buf);

        // at least four bytes are ready and can be written without further
        // checking
        if(count>=4)
        {
            REG(IR_UART_BASE, UTDR) = ringbuf_ReadByte(&sir_transmit_buf);
            REG(IR_UART_BASE, UTDR) = ringbuf_ReadByte(&sir_transmit_buf);
            REG(IR_UART_BASE, UTDR) = ringbuf_ReadByte(&sir_transmit_buf);
            REG(IR_UART_BASE, UTDR) = ringbuf_ReadByte(&sir_transmit_buf);
        }
        else
        {
            for(i=0;i<count;i++)
                REG(IR_UART_BASE, UTDR) = ringbuf_ReadByte(&sir_transmit_buf);

            // output finished, disable the interrupt
            SER_UART_DISABLE_TI(IR_UART_BASE);
        }
    }

#else
    {
        INT_MASK(INT_UART2);    // should not happen
        debug_String("ERROR: unexpected transmit interrupt occurred, "
                     " serial port 2 interrupt shut down.");
    }
#endif
}


/*
 * Function: sir_NewReceiveData_Q
 *  Purpose: Check if new data has been received.
 *
 * Parameters:
 *      Input: none
 *
 *    Returns: BOOL indicating whether new data has been received or not
 *
 * This function can be used to check if new received data is available in the
 * receive ringbuffer.
 */
BOOL sir_NewReceiveData_Q(void)
{
    return sir_new_receive_data;
}




/*
 * Function: sir_SetReceiveMode
 *  Purpose: Set the SIR port to receive mode.
 *
 * Parameters:
 *      Input: none
 *
 *    Returns: void
 *
 * If UART transmission is configured the function first waits for any ongoing
 * transmission to be finished. Then the transmitter interrupt and the transmit
 * unit are disabled.
 * In every case the receive FIFO is emptied to remove any reflected data from
```

Listing 38. Implementation of SIR-related functions

```
 * the transmit operation, then the status bits are cleared, and receive
 * interrupt and receiver are enabled.
 */
void sir_SetReceiveMode(void)
{
    UC8 dummy;

#ifdef SIR_UART_TRANSMISSION
    {
        int delay;

        // wait until transmission is finished. Not necessary if
        // software modulation is used

        // UART TBY flag not immediately set after transmitter enable.
        for(delay=0;delay<SIR_TRANSM_BUSY_DELAY;delay++);
    }
    while (SER_UART_TRANSM_BUSY_Q(IR_UART_BASE))
    {
        ;
    }

    // disable transmitter and transmitter interrupt
    SER_UART_DISABLE_T(IR_UART_BASE);
    SER_UART_DISABLE_TI(IR_UART_BASE);
#endif

    // make sure receiver FIFO doesn't contain any reflected data
    while(SER_UART_REC_NOT_EMPTY_Q(IR_UART_BASE))
    {
        dummy = REG(IR_UART_BASE, UTDR);
    }

    // clear status bits and then enable receiver interrupt and receiver
    SER_UART_CLEAR_STATUS_BITS_ALL(IR_UART_BASE);
    SER_UART_ENABLE_RI(IR_UART_BASE);
    SER_UART_ENABLE_R(IR_UART_BASE);
}


/*
 * Function: sir_SetTransmitMode
 *  Purpose: Set the SIR port to transmit mode.
 *
 * Parameters:
 *      Input: none
 *
 *    Returns: void
 *
 * First any remaining bytes in the FIFO are read and written to the receive
 * buffer, then receiver and receiver interrupt are disabled.
 * If UART transmission is configured the function then enables the transmit
 * interrupt and the transmit unit, otherwise it just clears the TXD2 pin.
 */
void sir_SetTransmitMode(void)
{

    // read any bytes left in the receiver FIFO
    // should cause an interrupt but for some reason doesn't do always
    while(SER_UART_REC_NOT_EMPTY_Q(IR_UART_BASE))
    {
        ringbuf_WriteByte(&sir_receive_buf, REG(IR_UART_BASE, UTDR));
    }

    // disable receiver and receiver interrupt
```

Listing 38. Implementation of SIR-related functions

```
    SER_UART_DISABLE_R(IR_UART_BASE);
    SER_UART_DISABLE_RI(IR_UART_BASE);

#ifdef SIR_UART_TRANSMISSION
    // clear status bits and then enable transmit interrupt and transmitter
    SER_UART_CLEAR_STATUS_BITS_ALL(IR_UART_BASE);
    SER_UART_ENABLE_TI(IR_UART_BASE);
    SER_UART_ENABLE_T(IR_UART_BASE);
#else
    // reset to zero before transmitting
    PPC_CLEAR_PIN(TXD2);
#endif
}


/*
 * Function: sir_RestartTransmUART
 *  Purpose: Restart SIR transmission using the UART.
 *
 * Parameters:
 *      Input: none
 *
 *    Returns: void
 *
 * The function restarts the SIR transmission, but doesn't change any other
 * setting or mode, i.e., it can be used to restart a transmission when new
 * data has become available and no mode change has been done before.
 * Note: this function only works for UART transmission mode !!
 */
void sir_RestartTransmUART(void)
{
    SER_UART_ENABLE_TI(IR_UART_BASE);
}


/*
 * Function: sir_EnqueueTransmData
 *  Purpose: Write data to the SIR transmission buffer.
 *
 * Parameters:
 *      Input: data     pointer to the data to be written to the transmit buffer.
 *             length   length of the buffer to be written.
 *
 *    Returns: int      the number of actually written bytes.
 *
 * The function first checks the available space in the ringbuffer and then
 * transfers as many bytes as possible from the given buffer to the transmit
 * ringbuffer. It returns the number of actually written bytes which can be less
 * then the supplied value if the transmit buffer doesn't have enough free
 * space.
 */
int sir_EnqueueTransmData(UC8* data, UC8 length)
{
    int i=0;
    int avail;
    UC8 cur;

    avail = ringbuf_GetSpace(&sir_transmit_buf);
    while(i<length && i<avail)
    {
        cur = *(data+i++);
        ringbuf_WriteByte(&sir_transmit_buf, cur);
    }
    return i;
}
```

Listing 38. Implementation of SIR-related functions

```
/*
 * Function: sir_EnqueueTransmByte
 *  Purpose: Write a single byte to the SIR transmission buffer.
 *
 * Parameters:
 *      Input: ch       byte to be written to the buffer
 *
 *    Returns: int      the number of actually written bytes.
 *
 * The function writes the given byte to the transmit buffer. Currently for
 * speed reasons it does no check of the available buffer space.
 */
void sir_EnqueueTransmByte(UC8 ch)
{
    ringbuf_WriteByte(&sir_transmit_buf, ch);
    return;
}


/*
 * Function: sir_EnqueueTransmByteMultiple
 *  Purpose: Write a byte to the SIR transmission buffer multiple times.
 *
 * Parameters:
 *      Input: val      the byte to write
 *             count    how oft the byte is to be written
 *
 *    Returns: int      the number of actually written bytes.
 *
 * The function first checks the available space in the ringbuffer and then
 * tries to write the given byte the given number of times to the transmit
 * ringbuffer. It returns the number of actually written bytes which can be
 * less then the supplied value if the transmit buffer doesn't have enough free
 * space.
 */
int sir_EnqueueTransmByteMultiple(UC8 val, int count)
{
    int i=0;
    int avail = ringbuf_GetSpace(&sir_transmit_buf);
    while(i< avail && i< count)
    {
        ringbuf_WriteByte(&sir_transmit_buf, val);
        i++;
    }

    return i;
}



/*
 * Function: sir_TrasnmitDataSW(void)
 *  Purpose: Start the transmission using software modulation
 *
 * Parameters:
 *      Input: none
 *
 *    Returns: void
 *
 * The function sets a short timeout for the timer match register and then
 * unmasks the timer interrupt chosen for the software modulation to start
 * the transmission after the set timeout.
 * Note: This function can only be used for software modulation transmission !
```

Listing 38. Implementation of SIR-related functions

```
 */
#ifndef SIR_UART_TRANSMISSION
void sir_TransmitDataSW(void)
{
    /* start transmission */
    if(!sir_sw_transmitting)
    {
        sir_sw_transmitting = TRUE;
        OSTIMER_INC_MATCH_REG(sir_ostimer_channel, 40);
        INT_UNMASK(sir_ostimer_int);
    }
}
#endif


/*
 * Function: sir_GetReceiveCount
 *  Purpose: Get the number of bytes available in the receive buffer.
 *
 * Parameters:
 *      Input: none
 *
 *    Returns: int      the number of available bytes.
 *
 * The function resets the flag indicating newly received data and then
 * returns the number of bytes currently in the receive ringbuffer.
 */
int sir_GetReceiveCount(void)
{
    sir_new_receive_data = FALSE;
    return ringbuf_GetCount(&sir_receive_buf);
}


/*
 * Function: sir_GetReceiveData
 *  Purpose: Read the data from the receive buffer
 *
 * Parameters:
 *      Input: len      the number of bytes to read from the receive buffer.
 *
 *      Input/Output:
 *             data     a pointer into a buffer to receive the bytes
 *
 *    Returns: int      the number of actually read bytes.
 *
 * The function reads the given number of bytes from the receive ringbuffer
 * and stores them in the supplied buffer. This buffer must have enough space
 * to receive all read bytes. The function returns the number of bytes read.
 */
int sir_GetReceiveData(UC8* data, int len)
{
    int i;
    for(i=0;i<len;i++)
    {
        *(data+i) = ringbuf_ReadByte(&sir_receive_buf);
    }
    return i;
}


/*
 * Function: sir_PrintDebugBuf
 *  Purpose: Write the contents of the software modulation debug buffer to
 *           stdout.
```

Listing 38. Implementation of SIR-related functions

```
 *
 * Parameters:
 *      Input: none
 *
 *    Returns: void
 *
 * The function reads the contents of the software modulation debug buffer and
 * prints them to stdout byte by byte. When all bytes have been written, stdout
 * is flushed to make sure the bytes are actually sent.
 */
void sir_PrintDebugBuf(void)
{
#ifdef SIR_SW_DEBUG_BUF

    int len, i;
    UC8 ch;

    len = ringbuf_GetCount(&sir_sw_debug_buf);
    for(i=0;i<len;i++)
    {
        ch= ringbuf_ReadByte(&sir_sw_debug_buf);
        putchar(ch);
    }
    fflush(stdout);
#endif
}
```

Listing 38. Implementation of SIR-related functions

A.3 Implementation of the FIR-Functions

```
/*
 *      Description:
 *          Code to support infrared communication in FIR mode on the Badge.
 *          Right now only an initialization function and two functions for
 *          polled transmitting and sending are implemented, functions for
 *          interrupt driven sending and transmitting, and mode changes can
 *          easily be added in analogy to the SIR code.
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#include <util/util_misc.h>
#include <util/util_ir.h>


/*
 * Function: fir_Init
 *  Purpose: Initialize serial port 2 for FIR mode
 *
 * Parameters:
 *      Input: receive_int_en   en-/disable the receive interrupt request
 *             transm_int_en    en-/disable the transmit interrupt request
 *             addr_match_en    en-/disable address match functionality
 *             addr_match_val   the address match value
 *
 *    Returns: void
```

Listing 39. Implementation of FIR-related functions

```
 *
 * The function initializes all control registers of the HSSP, clears
 * the sticky status bits, but doesn not yet enable the receiver and
 * transmitter.
 * TUS is set to 0 by default, thus transmit FIFO underruns cause the frame
 * to be finished. This can easily be changed by calling the according macros.
 * See the constants defined in util_ir.h to be passed as parameters.
 * Note: the function sets the RXD2 pin polarity to inverted to make FIR work
 *       on the Badge. This has to be considered when later using the PPC
 *       or the UART.
 */
void fir_Init(BOOL receive_int_en, BOOL transm_int_en, BOOL addr_match_en,
        UC8 addr_match_val)
{
    IR_HSSP_SHUTDOWN;

    // for some reason FIR on the Badge requires the RXD2 pin to be operated
    // in inverted mode
    IR_SET_POLARITY_RXD2_INVERTED;

    // set the mode pin for FIR
    GPIO_SET_OUTPUT(IRSD);
    GPIO_SET_PIN(IRSD);

    IR_HSSP_SET_ADDR_MATCH(addr_match_val);
    IR_HSSP_CLEAR_STATUS_BITS_ALL;
    REG(HSSP_BASE, HSCR0) = 0x1 | receive_int_en << 5 | transm_int_en << 6 |
                            addr_match_en << 7;
}


/*
 * Function: fir_GetBytePolled
 *  Purpose: Read a byte from serial port 2 in FIR mode (polling).
 *
 * Parameters:
 *      Input: none
 *
 *    Returns: the byte read from the HSSP receive FIFO
 *
 * The function waits in a busy loop until there is a byte available in the
 * HSSP receive FIFO as indicated by the flag RNE in status register 1. When
 * a byte is available it is read and returned.
 */
unsigned char fir_GetBytePolled(void)
{
    while(!IR_HSSP_REC_NOT_EMPTY_Q)
        ;

    return IR_HSSP_GET_BYTE_DIRECT;
}


/*
 * Function: fir_PutBytePolled
 *  Purpose: Write a byte to serial port 2 in FIR mode (polled).
 *
 * Parameters:
 *      Input: ch   the byte to be written
 *
 *    Returns: void
 *
 * The function waits in a busy loop until there is free space in the
 * HSSP transmit FIFO as indicated by the flag TNF in status register 1. When
 * at least one entry is free the byte is written to the FIFO.
```

Listing 39. Implementation of FIR-related functions

```
 */
void fir_PutBytePolled(unsigned char ch)
{
    while(!IR_HSSP_TRANSM_NOT_FULL_Q)
        ;

    IR_HSSP_PUT_BYTE_DIRECT(ch);
}
```

Listing 39. Implementation of FIR-related functions

Appendix B. Example Programs

This chapter presents a few example programs that are meant to clarify the use of some of the definitions, macros and functions that I described in Chapter 4. "Using some of the Peripherals" and Chapter 5. "Infrared Communication". The files containing the code presented in these two chapters have been compiled into a library which has been linked to the files presented in the following sections. An exception is the file util_sir.c (SIR mode infrared communication) as this can be configured to support sofware modulation or UART transmission and therefore has to be recompiled for each application.

B.1 GPIO and PPC - Maximum Frequency Pin Toggling

B.1.1 Standalone Version

```
/*
 *      Description:
 *          The program toggles a GPIO / PPC pin to find the highest
 *          possible toggling frequency.
 *          If GPIO is defined, the GPIO pin is used, otherwise the PPC
 *          pin (TXD2 - the IR TXD).
 *          Defining PPC_OPT causes an optimized loop for the PPC toggling
 *          to be used, as opposed to the compiler generated version.
 *          It uses an assembler function local_pos() to get the current
 *          location in the code and provides a function to relocate the
 *          toggling loop in the memory space from FLASH to SRAM (If
 *          RELOCATE is defined).
 *          This version is a standalone version and uses init.s to
 *          initialize the processor.
 *          Performance depends on if the instruction cache is enabled in
 *          init.s or not.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 */

#include <util/util_gpio.h>
#include <util/util_ppc.h>
#include <string.h>


// configuration settings

/* use GPIO or PPC */
//#define GPIO

/* use compiler generated or optimized PPC loop */
#define PPC_OPT

/* do relocation (FLASH->SRAM) or not (execute out of FLASH) */
#define RELOCATE

/* which GPIO pin to use */
#define GPIO_PIN GPIO6
```

Listing 40. GPIO/PPC pin toggling, standalone version

```
/* the location where in SRAM to relocate the toggle loop */
#define RAM_BASE 0x08001000



// return the current position in the code
extern int local_pos(void);

/* this function copies a number of bytes from an offset given by the
 * argument pos to a new memory location given by rambase and branches
 * there by modifying the stack
 */
void copy(int rambase, int pos)
{
    memcpy((unsigned char*)rambase, (unsigned char*)pos, 200);

    // get the starting address of the code to execute and write that
    // address into the stack position which upon exiting the function
    // is restored into the PC
    // the offset of 8 is caused by
    __asm
    {
        mov     r14, RAM_BASE
        str     r14, [r13, #+8]
    }
}


int main(void)
{
    int pos;

#ifdef GPIO
    // make GPIO pin 6 an output
    GPIO_SET_OUTPUT(GPIO_PIN);
    GPIO_CLEAR_PIN(GPIO_PIN);
#else
    // enable TXD2 as PPC output
    PPC_SET_OUTPUT(TXD2);
    PPC_CLEAR_PIN(TXD2);
#endif

#ifdef RELOCATE

    // relocate the toggle loop to SRAM

    // get the current location in the code
    pos = local_pos();

    //copy the toggling loop to a different location and branch there
    copy(RAM_BASE, pos+16);
#endif

    while(1)
    {
#ifdef GPIO
        // toggle the GPIO pin
        GPIO_SET_PIN(GPIO_PIN);
        GPIO_CLEAR_PIN(GPIO_PIN);
#else
        // toggle the PPC pin

    #ifdef PPC_OPT
```

Listing 40. GPIO/PPC pin toggling, standalone version

```
    // optimized version of the loop
    // first load r0 with the address of the PPC pin state register,
    // then in the loop set/clear the bit for TXD2
    __asm
    {
        MOV      r0,#0x60000
        ADD      r0,r0,#0x90000004

loop:
        ldr r2, [r0, #0]
        orr r2, r2, #0x4000
        str r2, [r0, #0]

        ldr r2, [r0, #0]
        bic r2, r2, #0x4000
        str r2, [r0, #0]
        b loop
    }

#else
        PPC_SET_PIN(TXD2);
        PPC_CLEAR_PIN(TXD2);
#endif  // PPC_OPT

#endif  // GPIO
    }
}
```

Listing 40. GPIO/PPC pin toggling, standalone version

B.1.2 Angel Version

```
/*
 *      Description:
 *          The program toggles a GPIO/PPC pin to find the highest
 *          possible toggling frequency.
 *          If GPIO is defined, the GPIO pin is used, otherwise the PPC
 *          pin (TXD2 - the IR TXD).
 *          Relocation doesn't make much sense here (compare the standalone
 *          version).
 *          In this version the cache can be en-/disabled by un-/commenting
 *          the define NO_CACHE.
 *          Defining PPC_OPT causes an optimized loop for the PPC toggling
 *          to be used, as opposed to the compiler generated version.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 */

#include <util/util_gpio.h>
#include <util/util_ppc.h>
#include <util/util_misc.h>
#include <string.h>



// configuration settings

/* use GPIO or PPC */
```

Listing 41. GPIO/PPC pin toggling, Angel version

```
//#define GPIO

/* enable or disable instruction cache */
//#define NO_CACHE

/* use compiler generated or optimized PPC loop */
#define PPC_OPT

/* which GPIO pin to use */
#define GPIO_PIN GPIO6


int main(void)
{

    misc_InitAngelFunctions();

#ifdef NO_CACHE
    misc_SysDisableICache();
#else
    misc_SysEnableICache();
#endif


#ifdef GPIO
    // make the defined GPIO pin an output
    GPIO_SET_OUTPUT(GPIO_PIN);
    GPIO_CLEAR_PIN(GPIO_PIN);
#else
    // enable TXD2 as PPC output
    PPC_SET_OUTPUT(TXD2);
    PPC_CLEAR_PIN(TXD2);
#endif


    // the toggle loop
    while(1)
    {
#ifdef GPIO
        // toggle the GPIO pin
        GPIO_SET_PIN(GPIO_PIN);
        GPIO_CLEAR_PIN(GPIO_PIN);
#else
        // toggle the PPC pin

#ifdef PPC_OPT

    // optimized version of the loop
    // first load r0 with the address of the PPC pin state register,
    // then in the loop set/clear the bit for TXD2
    __asm
    {
        MOV      r0,#0x60000
        ADD      r0,r0,#0x90000004

loop:
        ldr r2, [r0, #0]
        orr r2, r2, #0x4000
        str r2, [r0, #0]

        ldr r2, [r0, #0]
        bic r2, r2, #0x4000
        str r2, [r0, #0]
        b loop
    }
```

Listing 41. GPIO/PPC pin toggling, Angel version

```
#else
        PPC_SET_PIN(TXD2);
        PPC_CLEAR_PIN(TXD2);
#endif  // PPC_OPT

#endif  // GPIO
    }
}
```

Listing 41. GPIO/PPC pin toggling, Angel version

B.2 GPIO - Interrupt Latency

This program was described in interrupt section 4.3.4 - note that the results are not reproducible due to changes in the library done after the measurements (see section 4.3.4.1)

```
/*
 *      Description:
 *          Test code for exploring gpio interrupts on the Badge.
 *          The code configures GPIO pin 3 (and 6 for test purposes) as
 *          outputs and another one, currently pin 2, as an input with edge
 *          detection on rising edges. Then it sets pin 3 (which has to be
 *          connected to the input pin) to generate a rising edge. The time
 *          difference between this and the interrupt handler being executed
 *          is measured using the OS Timer Count Register. This is done 15
 *          times in a loop and can be executed with caches dis- or enabled
 *          and with printf-statements within the loop which severely affects
 *          the effectivity of the caches (#define WITH_PRINTF)
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>
#include    <math.h>

#include <util/util_gpio.h>
#include <util/util_interrupt.h>
#include <util/util_ostimer.h>

volatile int int_flag = 0;
volatile UI32 stop;
#define INT_PIN GPIO2
#define INT_CHANNEL INT_GPIO2


// Configuration

// choose the desired combination of caches by
// commenting/uncommenting the defines
//#define I_CACHE_ENABLED
//#define D_CACHE_ENABLED

// use or don’t use printf in the measurement loop
#define WITH_PRINTF
```

Listing 42. GPIO interrupts

```
// optionally output the interrupt handler table
//#define DEBUG_INT_TABLE




// get the current count of the OS Timer Count Register, clear the interrupt
// request flag and set a status variable to signal the end of the measurement
void GPIO_IntHandler(unsigned int ident, unsigned int data,
                     unsigned int empty_stack)
{
    stop = OSTIMER_GET_COUNT;
    if(ident>=2 && ident <= 9)
    {
        GPIO_CLEAR_EDGE_STATUS(1<<ident);
    }
    else
    {
        // only GPIO 2 to 9 freely available, no other GPIO interrupt should
        // occur
        INT_MASK(ident);
    }

    int_flag = 1;   // measurement finished
}



int main(void){

    UI32 start;
    int diff;
    UI32 results[20];
    int i,j=0;

    misc_InitAngelFunctions();  // necessary for cache en-/disable


    // setup the desired combination of caches
#ifdef I_CACHE_ENABLED
    misc_SysEnableICache();
    printf("\nInstruction cache enabled, ");
#else
    misc_SysDisableICache();
    printf("\nInstruction cache disabled, ");
#endif


#ifdef D_CACHE_ENABLED
    misc_SysEnableDCache();
    printf("Data cache enabled\n\n");
#else
    misc_SysDisableDCache();
    printf("Data cache disabled\n\n");
#endif


#ifdef DEBUG_INT_TABLE
    int_PrintHandlerTable();
#endif

    int_InstallHandler(INT_CHANNEL, GPIO_IntHandler);

#ifdef DEBUG_INT_TABLE
```

Listing 42. GPIO interrupts

```
    printf("Handler for interrupt GPIO is at 0x%X\n",(int)(GPIO_IntHandler));
    int_PrintHandlerTable();
#endif

    GPIO_CLEAR_EDGE_STATUS_ALL; // clear the GEDR bits
    GPIO_SET_OUTPUT(GPIO3 | GPIO6); // pin 3,6 as output
    GPIO_CLEAR_PIN(GPIO3 | GPIO6);  // clear pin 3,6

    GPIO_SET_INPUT(INT_PIN);    // interrupt pin as input

    // clear the GEDR bit int pin,3, 6 after clearing pin 3 and 6,
    // just to make sure
    GPIO_CLEAR_EDGE_STATUS(INT_PIN | GPIO3 | GPIO6);

    GPIO_ENABLE_RISING_EDGE(INT_PIN);  // rising edge detect for the int pin

    // not really necessary, but to make sure that there are no problems
    // with wrap around
    OSTIMER_SET_COUNT(0);


    INT_UNMASK(INT_CHANNEL);


    // actual measurement loop
    for(i=0;i<15;i++)
    {

#ifdef WITH_PRINTF
        printf("Set GPIO pin 3 to cause a rising edge on the int pin\n");
#endif

        start = OSTIMER_GET_COUNT;
        GPIO_SET_PIN(GPIO3);
        while(!int_flag)
            ;

        diff = stop - start;
        int_flag = 0;

#ifdef WITH_PRINTF
        printf("GPIO Interrupt occured.\n");
        printf("Interrupt delay: %d ticks = %fs\n\n", diff, diff / 3686400.0);
#else
        results[i] = diff;
#endif
        GPIO_CLEAR_PIN(GPIO3);
        GPIO_CLEAR_EDGE_STATUS(INT_PIN);

        // wait a little bit (a more elegant way would be to use a timer...)
        for(j=0;j<500000;j++);

    }

#ifndef WITH_PRINTF

    // output the results
    for(i=0;i<15;i++)
    {
        printf("Interrupt delay: %u ticks = %fs\n\n", results[i],
                results[i] / 3686400.0);
    }
#endif

    // clear up
```

Listing 42. GPIO interrupts

```
    INT_MASK(INT_CHANNEL);
    GPIO_DISABLE_RISING_EDGE(INT_PIN);

    return (0);
}
```

Listing 42. GPIO interrupts

B.3 Real Time Clock Example

```
/*
 *      Description:
 *          Test code for exploring interrupts BIT 30 & 31 on the Badge.
 *          This code is derived from the original OneHertz program done
 *          by Prof. G. Q. Maguire and Mat C. Hans. My version uses the
 *          macros and functions provided in my support code and corrects
 *          an error within the original code ( resetting the interrupt
 *          status flags AL and HZ was done using code of the form
 *          “status_register |= flag_to_reset” which resets all flags, not
 *          only the intended one !)
 *          The program configures the real time clock and installs an
 *          interrupt handler which increments a counter every second. This
 *          is detected by the main program and a message is output each
 *          time. In addition the alarm is set to 5 sec and resets the
 *          counter to 0 when it goes off.
 *          Note the difference between the one- and the two-handler version:
 *          When the alarm occurs, both interrupt bits are set !
 *          RTC_Handler:  first the one hz interrupt is serviced,
 *            incrementing the counter, then the alarm interrupt is serviced
 *            which resets the counter. This is detected by the main loop
 *            and “counter 0” is output once.
 *          Two handlers: the alarm int has higher priority, thus is serviced
 *            first -> counter reset. Then, before the main loop has a chance
 *            to detect that, the one hz int is serviced, incrementing the
 *            counter -> the “counter 0” state is lost.
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 */


#include    <stdio.h>

#include <util/util_interrupt.h>
#include <util/util_realtime.h>

// separate int handlers for one hertz and alarm or one combined handler
//#define TWO_INTHANDLER
#undef TWO_INTHANDLER


volatile int counter = 0, bitset = -1;

#ifdef TWO_INTHANDLER
/*
 * Interrupt handler for One Hz. VERY IMPORTANT: reset HZ bit
 */
void OneHZ_IntHandler(unsigned int ident, unsigned int data, unsigned int empty_stack)
{
```

Listing 43. Realtime clock

```
    if (REALTIME_GET_STATUS_ONE_HZ)
    {
        // reset the interrupt flag and increment the counter
        REALTIME_CLEAR_ONE_HZ_INT;
        counter++;
        bitset = REALTIME_HZ;
    }
    else
    {
        // shouldn't be here... remove RTC interrupts
        bitset = -2;
        REALTIME_DISABLE_ONE_HZ_INT;
    }
}


/*
 * Interrupt handler for alarm. VERY IMPORTANT: reset AL bit
 */
void Alarm_IntHandler(unsigned int ident, unsigned int data, unsigned int empty_stack)
{
    if (REALTIME_GET_STATUS_ALARM)
    {
        // reset the interrupt flag and reset the counter to 0
        REALTIME_CLEAR_ALARM_INT;
        counter=0;
        bitset = REALTIME_AL;
    }
    else
    {
        // shouldn't be here... remove RTC interrupts
        bitset = -3;
        REALTIME_DISABLE_ALARM_INT;
    }
}

#else

/*
 * Interrupt handler for alarm and Hz
 */
void RTC_IntHandler(unsigned int ident, unsigned int data, unsigned int empty_stack)
{
    if(REALTIME_GET_STATUS_ONE_HZ || REALTIME_GET_STATUS_ALARM)
    {
        if (REALTIME_GET_STATUS_ONE_HZ)
        {
            REALTIME_CLEAR_ONE_HZ_INT;
            counter++;
            bitset = REALTIME_HZ;
        }

        if (REALTIME_GET_STATUS_ALARM)
        {
            REALTIME_CLEAR_ALARM_INT;
            counter = 0;
            bitset = REALTIME_AL;
        }
    }
    else
    {

        // shouldn't be here... remove RTC interrupts
        bitset = -4;
        REALTIME_RESET_FLAGS;
```

Listing 43. Realtime clock

```
    }

}
#endif



int main(void)
{
    int lcount = counter;

    fprintf(stdout, "\n\nStart of program which explores RTC interrupts\n");
    fflush(stdout);

    fprintf(stdout, "\nExamine the list of interrupt handlers\n");

    fprintf(stdout,"** Original interrupt function handler array:\n");
    int_PrintHandlerTable();

#ifdef TWO_INTHANDLER
    fprintf(stdout,"\n\nRedirect RTC interrupts:\n");
    fprintf(stdout,"BIT30: OneHZ handler 0x%X\n",(int)(&OneHZ_IntHandler));
    fprintf(stdout,"BIT31: Alarm handler 0x%X\n",(int)(&Alarm_IntHandler));

    int_InstallHandler(30, &OneHZ_IntHandler);
    int_InstallHandler(31, &Alarm_IntHandler);

#else
    fprintf(stdout,"\n\nRedirect RTC interrupts:\n");
    fprintf(stdout,"BIT30: RTC handler 0x%X\n",(int)(&RTC_IntHandler));
    fprintf(stdout,"BIT31: RTC handler 0x%X\n",(int)(&RTC_IntHandler));

    int_InstallHandler(30, &RTC_IntHandler);
    int_InstallHandler(31, &RTC_IntHandler);

#endif

    fprintf(stdout,"\n\n** Modified interrupt function handler array:\n");
    int_PrintHandlerTable();

    fprintf(stdout, "\n\nCurrent state:\n");
    fprintf(stdout,"*RTSR = 0x%X, *RTTR = 0x%X, *RCNR = 0x%X, "
         "*RTAR = 0x%X, counter %d, bitset = %d\n", REALTIME_GET_STATUS,
         REG(REALTIME_BASE, RTTR), REALTIME_GET_COUNT,
         REALTIME_GET_ALARM, counter, bitset);

    fprintf(stdout,"\nSetting up RTC and enabling interrupts\n");
    fflush(stdout);

    // The RTTR is programmed by the user to select the frequency of the "1 Hz"
    // clock. If this register is not programmed and left at it's reset value
    // (all zeros) then the "1 Hz" clock will actually be running at 32 768 Hz.
    REALTIME_SET_TRIM(0x7FFF, 0);

    // the RTC incorporates a 32 bit alarm register (RTAR). The RTAR
    // may be programmed with a value to be compared against the counter.
    // On each rising edge of the 1 hz clock, the counter is incremented
    // and then compared to the RTAR. If the values match, then a status
    // bit is set. This status bit is also routed to the interrupt controller
    REALTIME_INC_ALARM(5);  // interrupt in 5 seconds


    // reset the flags and enable the two interrupts
    REALTIME_RESET_FLAGS;
```

Listing 43. Realtime clock

```
    REALTIME_ENABLE_ONE_HZ_ALARM_INT;


    fprintf(stdout,"*RTSR = 0x%X, *RTTR = 0x%X, *RCNR = 0x%X, "
         "*RTAR = 0x%X, counter %d, bitset = %d\n", REALTIME_GET_STATUS,
         REG(REALTIME_BASE, RTTR), REALTIME_GET_COUNT,
         REALTIME_GET_ALARM, counter, bitset);


    fprintf(stdout,"\nEntering loop...\n");fflush(stdout);
    for (;;)
    {

        if (lcount!=counter)
        {
            fprintf(stdout,"RECEIVED irq  *RCNR 0x%X, counter %d, bitset = %d\n",
                    REALTIME_GET_COUNT, counter, bitset);
            lcount = counter;
            bitset = 0;

#if 1
            if (counter>10)
            {
                // stop the interrupts, and break
                REALTIME_RESET_FLAGS;

//                *(unsigned int *)RTSR = 0x0; //SET_HZ | SET_AL;
                break;
            }
#endif
        }
    }

    fprintf(stdout,"end of program\n");
    return (0);
}
```

Listing 43. Realtime clock

B.4 OS Timer Examples

B.4.1 Basic Use of the Macros - Timer Controlled Pin Toggling

```
/*
 *      Description:
 *          Test code for exploring the operating system timers on the Badge.
 *          The main loop reads a value from the serial port UART 1 to set
 *          the increment for the OS timer. The OS interrupt routine then
 *          toggles a GPIO 6 pin with a frequency depending on this timeout.
 *
 *          In the current configuration the terminal program needs to be
 *          set to 115200 baud, 8N1.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
```

Listing 44. Timer controlled pin toggling

```
 */

#include    <stdio.h>

#include <ctype.h>
#include <string.h>

#include <util/util_serial.h>
#include <util/util_interrupt.h>
#include <util/util_misc.h>
#include <util/util_ostimer.h>
#include <util/util_gpio.h>


#define BUF_SIZE 20

// the GPIO pin to toggle
#define TOGGLE_PIN GPIO6


int inc;
int int_count=0;

// ostimer interrupt handler
void OSTIMER0_IntHandler(unsigned int ident, unsigned int data,
                         unsigned int empty_stack)
{
    static unsigned int state=0;

    // set the increment for the match register
    OSTIMER_INC_MATCH_REG(OSTIMER_CHANNEL_0, inc);

    // depending on the current state set or clear the pin
    if(state)
    {
        GPIO_CLEAR_PIN(TOGGLE_PIN);
        state=0;
    }
    else
    {
        GPIO_SET_PIN(TOGGLE_PIN);
        state=1;
    }

    // clear the interrupt status flag
    OSTIMER_RESET_INT(OSTIMER_CHANNEL_0);
}


int main(void){

    unsigned char ch;
    unsigned char buf[BUF_SIZE];
    unsigned int bufcount = 0;
    unsigned int new_val;


    // install the ostimer handler and enable the interrupt
    OSTIMER_ENABLE_INT(OSTIMER_CHANNEL_0);
    int_InstallHandler(INT_OSTIMER_0, OSTIMER0_IntHandler);


    // enable TOGGLE_PIN as output
    GPIO_SET_OUTPUT(TOGGLE_PIN);
    GPIO_CLEAR_PIN(TOGGLE_PIN);
```

Listing 44. Timer controlled pin toggling

```
    // set up UART1 to read the timer values
    ser_UartInit(UART1_BASE, SER_INIT_BAUD_115200, SER_INIT_PARITY_DIS,
                  SER_INIT_PARITY_ODD, SER_INIT_ONE_STOP_BIT,
                  SER_INIT_DATA_SIZE_8, SER_INIT_RECEIVE_INT_DIS,
                  SER_INIT_TRANSMIT_INT_DIS);

    SER_UART_ENABLE_TR(UART1_BASE);

    printf("\nUsage: set up your terminal program for 115200 baud, 8N1.\n");
    printf("       Then enter decimal integer values to be used as OS Timer "
           "timeouts.\n");
    printf("       Space or enter after the value commit it "
           "(characters other than digits are ignored).\n\n");

    while(1)
    {
        ch=ser_UartGetBytePolled(UART1_BASE);
        ser_UartPutBytePolled(UART1_BASE, ch);

        if(isspace(ch) && bufcount>0)
        {
            new_val = atoi((char*)buf);
            memset(buf, 0x0, BUF_SIZE);
            bufcount=0;
            printf("new increment: %i\n", new_val);
            inc = new_val;

            OSTIMER_INC_MATCH_REG(OSTIMER_CHANNEL_0, inc);
            INT_UNMASK(INT_OSTIMER_0);
        }
        else if(isdigit(ch))
        {
            buf[bufcount++] = ch;
        }
    }
    return (0);
}
```

Listing 44. Timer controlled pin toggling

B.4.2 Use of the Software Timer Functions

```
/*
 *      Description:
 *          Demo code to show the use of the OS Timer functions.
 *          First two timers are initialized and then started, the first
 *          to go off every second, the second one to go off every half
 *          second. After five seconds the first timer is removed and
 *          lateron reinserted when the second timer has expired a certain
 *          number of times.
 *          Finally the programm waits 5 seconds and then outputs the
 *          contents of the debug buffer to check the timer sequence.
 *
 *          Note: the stdout output does not always show the correct sequence
 *                (1-2-2-1-2-2-1-2-2... sometimes something like
 *                 2-2-1-2-1-2-2-2-1-2-1-2-2 ... can occur). This depends on
 *                 where the checking loop is when the timer goes off. The
 *                 debug buffer which is output at the end of the program
```

Listing 45. Software timers

```
 *                  run should however always contain the correct sequence.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#include <stdlib.h>
#include <stdio.h>

#include “util/util_ostimer.h”
#include “util/util_serial.h”

// timeouts and which OS Timer channels to use
#define TIMER_1_TIMEOUT     1*SEC
#define TIMER_2_TIMEOUT     1*SEC/2
#define LIST_CHANNEL        OSTIMER_CHANNEL_0
#define WAIT_CHANNEL        OSTIMER_CHANNEL_1

// to collect debug output during timer callback execution
// (callbacks are queued functions and therefore can’t be debugged
// using the debugger)
RingBuffer DebugBuf;

// timer callbacks and variables to be set in the callbacks
void timer_1_expired(unsigned long data);
void timer_2_expired(unsigned long data);
int counter = 0;
int counter2 = 0;

// required for setting the new timeout
extern long ostimer_ticks;
//extern int ostimer_blocked;


// the two timers
struct timer_list timer_1;
struct timer_list timer_2;


// start the given timer with specified timeout and callback function
void start_timer( struct timer_list *ptimer, int timeout, int data,
                        TIMER_CALLBACK callback)
{
    ostimer_DelTimer( ptimer);

    ptimer->data = (unsigned long) data;
    ptimer->function = callback;

    ptimer->expires = ostimer_ticks + timeout;

    ostimer_AddTimer( ptimer);
}


// wrapper to start timer 1
void start_timer_1(int timeout)
{
    start_timer( &timer_1, timeout, (unsigned long) 0,
              timer_1_expired);
}


// wrapper to start timer 2
```

Listing 45. Software timers

```
void start_timer_2(int timeout)
{
    start_timer( &timer_2, timeout, (unsigned long) 0,
              timer_2_expired);
}


// callback function to be executed when timer 1 expires
// just increment the counter and restart the timer
void timer_1_expired(unsigned long data)
{
    ringbuf_WriteByte(&DebugBuf, '1');
    counter++;
    start_timer_1(TIMER_1_TIMEOUT);
    ringbuf_WriteByte(&DebugBuf, '2');
}


// callback function to be executed when timer 2 expires
// just increment the counter and restart the timer
void timer_2_expired(unsigned long data)
{
    ringbuf_WriteByte(&DebugBuf, '-');
    counter2++;
    start_timer_2(TIMER_2_TIMEOUT);
    ringbuf_WriteByte(&DebugBuf, '|');
}


int main(void)
{
    int lcount=0;
    int lcount2=0;
    int to1, to2;
    unsigned char* buf;
    BOOL cond = FALSE;

    to1 = to2 = FALSE;


    // not required as executed by ringbuf_Init() and ostimer_InitListInt
    // anyway, but just to remind that it should be put at the beginning
    // of application programs if there is a chance that they use any of
    // the Angel hooks.
    misc_InitAngelFunctions();

    if(ringbuf_Init(&DebugBuf, 2000) == -1)
    {
        printf("Error: not enough memory for debug buffer\n");
        exit(1);
    }

    ringbuf_WriteByte(&DebugBuf, 'i');
    printf("init\n");

    // init the software timer data structures
    ostimer_InitListInt(LIST_CHANNEL);
    ostimer_InitTimer(&timer_1);
    ostimer_InitTimer(&timer_2);

    // init a default timer for later use
    if(ostimer_InitChannelDefault(WAIT_CHANNEL, FALSE) ==-1)
    {
        printf("Error: timer channel already used.");
        exit(1);
```

Listing 45. Software timers

```
}


// start the two timers
start_timer_1(TIMER_1_TIMEOUT);
start_timer_2(TIMER_2_TIMEOUT);


while( !to1 || !to2 )
{
    if(lcount != counter)
    {
        lcount = counter;
        printf("timer_1 gone off\n");
        fflush(stdout);
        if(counter == 5)
        {
            ostimer_DelTimer(&timer_1);
            printf("timer_1 deleted\n");
            to1 = TRUE;
        }
        if(counter == 10)
        {
            ostimer_DelTimer(&timer_1);
            printf("timer_1 deleted\n");
            to1 = TRUE;
        }
    }
    if(lcount2 != counter2)
    {
        lcount2 = counter2;
        printf("timer_2 gone off\n");
        fflush(stdout);
        if(counter2== 15)
        {
            start_timer_1(TIMER_1_TIMEOUT);
            printf("timer_1 reinserted\n");
            to1 = FALSE;
        }

        if(counter2== 20)
        {
            ostimer_DelTimer(&timer_2);
            printf("timer_2 deleted\n");
            to2 = TRUE;
        }
    }

}

printf("Now 5 seconds delay, then exit.\n");
ostimer_WaitTime(WAIT_CHANNEL, 5, 's', &cond);

// clean up
ostimer_RemoveChannel(WAIT_CHANNEL);
ostimer_RemoveChannel(LIST_CHANNEL);
ostimer_DelTimer(&timer_1);
ostimer_DelTimer(&timer_2);

printf("\nExit, contents of the debug buffer:\n");
buf = (unsigned char*)malloc(2000);
if(buf)
{
    to1 = ringbuf_GetCount(&DebugBuf);
    ringbuf_ReadBuf(&DebugBuf, buf, to1);
```

Listing 45. Software timers

```
        buf[to1] = 0x0;
        printf("\n%s\n", buf);
    }
    else
        printf("no buffer");
}
```

Listing 45. Software timers

B.5 Serial Communication Example (UART)

B.5.1 Echo

```
/*
 *      Description:
 *          Simple echo program to demonstrate usage of the SA-1100
 *          UARTs.
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */


#include <stdlib.h>
#include <stdio.h>
#include <util/util_serial.h>



int main(void)
{
    char text[] = "\n\rBegin echo operation.\n\r";

    // output to stdout
    printf("\nInitializing UART 1\n");

    // initalize the UART ...
    ser_UartInit(UART1_BASE, SER_INIT_BAUD_115200, SER_INIT_PARITY_DIS,
                  SER_INIT_PARITY_ODD, SER_INIT_ONE_STOP_BIT,
                  SER_INIT_DATA_SIZE_8, SER_INIT_RECEIVE_INT_DIS,
                  SER_INIT_TRANSMIT_INT_DIS);

    // ... and enable transmitter and receiver
    SER_UART_ENABLE_TR(UART1_BASE);

    // output startup message on UART 1
    ser_UartPutStringPolled(UART1_BASE, (UC8*)text);

    while (1)
    {
        unsigned char ch;

        ch = ser_UartGetBytePolled(UART1_BASE);
        ser_UartPutBytePolled(UART1_BASE, ch);
    }
}
```

Listing 46. Basic echo program

B.6 IR SIR-Mode Examples

B.6.1 SIR Receiver in Polled Mode

```
/*
 *      Description:
 *          Basic example code for exploring the IRDA port on the Badge.
 *          Receiver part in slow infrared mode, bytes received via the
 *          infrared port are sent out on UART 1.
 *          For receiving UART 2 is simply polled.
 *
 *          Note: the program expects the file util_ir.c to be compiled
 *                with the symbol SIR_UART_TRANSMISSION defined !
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#include <util/util_serial.h>
#include <util/util_ir.h>

int main(void)
{

    unsigned char ch;

    // setup serial port 2 in 9600 baud SIR mode
    sir_Init(SER_INIT_BAUD_9600);
    SER_UART_ENABLE_R(IR_UART_BASE);

    // setup serial port 1 as 115kb UART and enable the transmitter
    ser_UartInitDefault(UART1_BASE, SER_INIT_BAUD_115200);
    SER_UART_ENABLE_T(UART1_BASE);

    ser_UartPutStringPolled(UART1_BASE, (UC8*)”begin \n\r”);

    while(1)
    {
        // read a character from serial port 2 in polling mode
        ch=ser_UartGetBytePolled(IR_UART_BASE);

        // SIR frames with no pulses are decoded as all ones,
        // don’t print them
        if(ch==0xff)
        {
            ser_UartPutBytePolled(UART1_BASE, ‘~’);
        }
        else
        {
            // write the character out to UART 1
            ser_UartPutBytePolled(UART1_BASE, ch);
        }
    }

    return (0);
}
```

Listing 47. SIR-receiver in polled mode

B.6.2 SIR Receiver in Interrupt Mode

```
/*
 *      Description:
 *          Basic example code for exploring the IRDA port on the Badge.
 *          The program initializes serial port 2 for 9600 baud SIR mode,
 *          then reads characters from the port in interrupt mode and
 *          prints them to stdout using printf.
 *
 *          Note: the program expects the file util_ir.c to be compiled
 *                with the symbol SIR_UART_TRANSMISSION defined  !!
 *
 *
 *      --Christoph Wolf
 *        chwolf@it.kth.se
 *
 */

#include <string.h>
#include <util/util_serial.h>
#include <util/util_ir.h>
#include <util/util_misc.h>

#define BUFLEN 100

int main(void)
{
    char readbuf[BUFLEN];
    int count;

    // initialize serial port 2 as 9600 baud UART using UART mode
    sir_Init(SER_INIT_BAUD_9600);

    // enable the UART2 interrupt and switch to receive mode
    INT_UNMASK(INT_UART2);
    sir_SetReceiveMode();


    while(1)
    {
        // wait for data on serial port 2
        while(!sir_NewReceiveData_Q())
            ;

        // get the number of bytes in the receive buffer
        count = sir_GetReceiveCount();

        while(count>0)
        {
            memset(readbuf, 0x0, BUFLEN);
            count -= sir_GetReceiveData((UC8*)readbuf, MIN(BUFLEN, count));
            printf(readbuf);
            fflush(stdout);
        }
    }

}
```

Listing 48. SIR-receiver in interrupt mode

B.6.3 SIR Transmitter Using UART2 in Polled Mode

```
/*
 *      Description:
 *          Basic example code for exploring the IRDA port on the Badge.
 *          The program initializes serial port 2 for 9600 baud SIR mode
 *          and transmits five lines consisting of the characters A-Z in
 *          polling mode.
 *
 *          Note: the program expects the file util_ir.c to be compiled
 *                with the symbol SIR_UART_TRANSMISSION defined and works
 *                only on Badges with the revised StrongARM (i.e. without
 *                the SIR bug of the original StrongARM) !!
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#include <util/util_serial.h>
#include <util/util_ir.h>


int main(void)
{
    char ch;
    int i=0;

    // initialize serial port 2 as 9600 baud UART using SIR modulation
    sir_Init(SER_INIT_BAUD_9600);
    SER_UART_ENABLE_T(IR_UART_BASE);

    while(i++<5)
    {
        ch='A';
        while(ch<='Z')
        {
            ser_UartPutBytePolled(IR_UART_BASE, ch++);
        }
        ser_UartPutBytePolled(IR_UART_BASE, 0x0a);
        ser_UartPutBytePolled(IR_UART_BASE, 0x0d);

    }

}
```

Listing 49. SIR-transmitter using the SA-1100 UART in polled mode

B.6.4 SIR Transmitter Using UART2 in Interrupt Mode

```
/*
 *      Description:
 *          Basic example code for exploring the IRDA port on the Badge.
 *          The program initializes serial port 2 for 9600 baud SIR mode
 *          and transmits five lines consisting of the characters A-Z in
 *          interrupt mode.
 *
 *          Note: the program expects the file util_ir.c to be compiled
 *                with the symbol SIR_UART_TRANSMISSION defined and works
```

Listing 50. SIR-transmitter using the SA-1100 UART in interrupt mode

```
 *                only on Badges with the revised StrongARM (i.e. without
 *                the SIR bug of the original StrongARM) !!
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#include <string.h>
#include <util/util_serial.h>
#include <util/util_ir.h>


int main(void)
{
    char string[] = “example transmission of a string.\n”;

    // initialize serial port 2 as 9600 baud UART using UART mode
    sir_Init(SER_INIT_BAUD_9600);

    // unmask serial port 2 interrupt and switch to transmit mode
    INT_UNMASK(INT_UART2);
    sir_SetTransmitMode();


    while(i++<5)
    {
        ch=’A’;

        // write the bytes to the transmit buffer ...
        while(ch<=’Z’)
        {
            sir_EnqueueTransmByte(ch++);
        }
        sir_EnqueueTransmByte(0x0a);
        sir_EnqueueTransmByte(0x0d);

        // ... and start transmission
        sir_RestartTransmUART();
    }

    sir_EnqueueTransmData((UC8*)string, strlen(string));
    sir_RestartTransmUART();

}
```

Listing 50. SIR-transmitter using the SA-1100 UART in interrupt mode

B.6.5 SIR Operation Using Software Modulation for Transmission

```
/*
 *      Description:
 *          Test code for exploring the IRDA port on the Badge.
 *          This program outputs the characters A-Z in a loop
 *          using SIR mode. As the on-chip SIR-mode modulator is buggy
 *          the program does software modulation using the os timer
 *          instead.
 *
 *          Note: it expects the file util_ir.c to be compiled with
 *                the flag SIR_UART_TRANSMISSION  NOT  defined ! (i.e.
 *                the #define statement has to be commented out)
```

Listing 51. SIR-transmitter using software modulation

```
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#include    <stdlib.h>

#include <util/util_serial.h>
#include <util/util_ir.h>
#include <util/util_debug.h>

int main(void) {
    char ch;
    int i,j=0;

    misc_InitAngelFunctions();
    debug_Init();

    // init serial port 2 as 9600b UART
    sir_Init(SIR_OSTIMER_CHANNEL);

    // set port to transmission mode
    sir_SetTransmitMode();

    while(j++<20)
    {
        ch='A';
        while(ch<='Z')
        {
            if(ringbuf_GetSpace(&sir_transmit_buf))
            {
                sir_EnqueueTransmByte(ch++);
            }
        }

        // add a CR/LF combination
        if(ringbuf_GetSpace(&sir_transmit_buf))
        {
            sir_EnqueueTransmByte(0xa);
        }
        if(ringbuf_GetSpace(&sir_transmit_buf))
        {
            sir_EnqueueTransmByte(0xd);
        }

        /* start transmission */
        sir_TransmitDataSW();

        /*
         * After transmission has started, no new bytes should be written
         * to the ring buffer. If this is done, the buffer contains still
         * valid data, all the bytes read-in in the SIR modulation routine
         * are correct, but transmission gets out of synchronization.
         * I was not able to determine the reason, a workaround is to wait
         * until transmission has finished before writing new data to the
         * buffer.
         */
        while(sir_SWTransmitting_Q())
            ;

        sir_PrintDebugBuf();
    }
}
```

Listing 51. SIR-transmitter using software modulation

B.7 IR FIR-Mode Examples

B.7.1 FIR Receiver in Polled Mode

```
/*
 *      Description:
 *          Simple example code for exploring the FIR mode on the Badge.
 *          The program reads bytes from serial port 2 in FIR mode and
 *          echos them on UART1.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#include <util/util_serial.h>
#include <util/util_misc.h>
#include <util/util_ir.h>

int main(void){

    UC8 ch;

    // set up serial port 2 in HSSP mode, interrupts disabled,
    // address matching enabled for address 0xfe
    fir_Init(HSSP_INIT_RECEIVE_INT_DIS, HSSP_INIT_TRANSMIT_INT_DIS,
             HSSP_INIT_ADDR_MATCH_EN, 0xfe);


    IR_HSSP_ENABLE_R;

    // set up serial port 1 as 115kb UART
    ser_UartInitDefault(UART1_BASE, SER_INIT_BAUD_115200);
    SER_UART_ENABLE_T(UART1_BASE);

    while(1)
    {
        ch=fir_GetBytePolled();
        if(ch!=0xff && ch!=0xfe) // don't print address (has to be adjusted if
        {                        // different address is used)
            ser_UartPutBytePolled(UART1_BASE, ch);
        }
    }

    return 0;

}
```

Listing 52. FIR-receiver in polled mode

B.7.2 FIR Transmitter in Polled Mode

```
/*
 *      Description:
 *          Simple example code for exploring the FIR mode on the Badge.
 *          The program sends sequences of 8 characters via serial port 2
 *          in FIR mode. After eight sequences a CR/LF pair is sent.
 *          Varying the first byte allows to experiment with the address
 *          match functionality.
```

Listing 53. FIR-transmitter in polled mode

```
 *          In this setup the TUS flag is cleared, hence a transmit FIFO
 *          underrun causes the current frame to be finished.
 *
 *
 *      --Christoph Wolf
 *         chwolf@it.kth.se
 *
 */

#include <util/util_ir.h>

int main(void){

    int i;
    int c=0;

    // init serial port 2 in FIR mode with receive and transmit interrupts,
    // and address match functionality disabled
    fir_Init(HSSP_INIT_RECEIVE_INT_DIS, HSSP_INIT_TRANSMIT_INT_DIS,
             HSSP_INIT_ADDR_MATCH_DIS, 0);

    // send groups of eight characters, after eight groups add CR/LF
    // and take a short pause
    while(1)
    {
        c++;
        IR_HSSP_ENABLE_T;
//      IR_HSSP_PUT_BYTE_DIRECT(0xef);  // unicast address
        IR_HSSP_PUT_BYTE_DIRECT(0xff);  // broadcast address
        IR_HSSP_PUT_BYTE_DIRECT('A');
        IR_HSSP_PUT_BYTE_DIRECT('B');
        IR_HSSP_PUT_BYTE_DIRECT('C');
        IR_HSSP_PUT_BYTE_DIRECT('D');
        IR_HSSP_PUT_BYTE_DIRECT('E');
        IR_HSSP_PUT_BYTE_DIRECT('F');
        IR_HSSP_PUT_BYTE_DIRECT('G');
        IR_HSSP_PUT_BYTE_DIRECT('H');
        if(c==8)
        {
            IR_HSSP_PUT_BYTE_DIRECT(0xa);
            IR_HSSP_PUT_BYTE_DIRECT(0xd);
            c=0;
        }
        i=0;
        while(i<50000)
            i++;

        IR_HSSP_DISABLE_T;

        i=0;
        while(i<50000000)
            i++;
    }

}
```

Listing 53. FIR-transmitter in polled mode

Appendix C. IrDA

This chapter contains the source code of the device driver files irport.c, wrapper.h and wrapper.c for the IrDA stack as described in section 9.4.4. and a sample debug log that shows the events and actions taking place when the IrDA stack is initialized and a connection to an infrared access point is established.

C.1 Implementation of irport.c

```
 1 /*********************************************************************
 2  *
 3  * Filename:      irport.c
 4  * Version:   1.0
 5  * Description:   Half duplex serial port SIR driver for IrDA.
 6  * Status:    Experimental.
 7  * Author:    Dag Brattli <dagb@cs.uit.no>
 8  * Created at:    Sun Aug  3 13:49:59 1997
 9  * Modified at:   Tue Jun  1 10:02:42 1999
10  * Modified by:   Dag Brattli <dagb@cs.uit.no>
11  * Sources:   serial.c by Linus Torvalds
12  *
13  *     Copyright (c) 1997, 1998, 1999 Dag Brattli, All Rights Reserved.
14  *
15  *     This program is free software; you can redistribute it and/or
16  *     modify it under the terms of the GNU General Public License as
17  *     published by the Free Software Foundation; either version 2 of
18  *     the License, or (at your option) any later version.
19  *
20  *     This program is distributed in the hope that it will be useful,
21  *     but WITHOUT ANY WARRANTY; without even the implied warranty of
22  *     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
23  *     GNU General Public License for more details.
24  *
25  *     You should have received a copy of the GNU General Public License
26  *     along with this program; if not, write to the Free Software
27  *     Foundation, Inc., 59 Temple Place, Suite 330, Boston,
28  *     MA 02111-1307 USA
29  *
30  *     This driver is ment to be a small half duplex serial driver to be
31  *     used for IR-chipsets that has a UART (16550) compatibility mode.
32  *     Eventually it will replace irtty, because of irtty has some
33  *     problems that is hard to get around when we don't have control
34  *     over the serial driver. This driver may also be used by FIR
35  *     drivers to handle SIR mode for them.
36  *
37  *
38  * Modified by: Christoph Wolf <chwolf@it.kth.se> for the
39  *                  SmartBadge IrDA-Project, Nov. 2000
40  *
41  *
42  ********************************************************************/
43
44 // SA1100 must be defined if the original (digital) SA1100 is used to get
45 // the alternative SIR transmitting functionality (software modulation)
46 //#define SA1100
47
48
49 #include <stdlib.h>
```

Listing 54. Listing of irport.c

```
 50 #include <stdio.h>
 51 #include <string.h>
 52 #include <irda/errno.h>
 53
 54 #include <irda/spinlock.h>
 55 #include <util/util_misc.h>
 56 #include <util/util_interrupt.h>
 57 #include <util/util_serial.h>
 58 #include <util/util_ir.h>
 59 #include <util/util_ppc.h>
 60 #include <util/util_ostimer.h>
 61 #include <util/util_debug.h>
 62
 63 #include <irda/irda.h>
 64 #include <irda/wrapper.h>
 65 #include <irda/irport.h>
 66 #include <irda/irlap_frame.h>
 67
 68
 69 //#define DEBUG_IRPORT
 70 #ifdef DEBUG_IRPORT
 71     #define DEBUG_STRING(c) debug_String(c)
 72     #define DEBUG_PUT_BYTE(c) debug_PutByte(c)
 73     #define DEBUG_PUT_BYTE_HEX(c) debug_PutByteHex(c)
 74     #define DEBUG_PUT_BYTE_DIRECT(c) debug_PutByteDirect(c)
 75     #define DEBUG_PUT_BYTE_POLLED(c)    debug_PutBytePolled(c)
 76 #else
 77     #define DEBUG_STRING(c)
 78     #define DEBUG_PUT_BYTE(c)
 79     #define DEBUG_PUT_BYTE_HEX(c)
 80     #define DEBUG_PUT_BYTE_DIRECT(c)
 81     #define DEBUG_PUT_BYTE_POLLED(c)
 82 #endif
 83
 84
 85 #define IO_EXTENT 8
 86
 87 static unsigned int qos_mtt_bits = 0x03;
 88
 89 // the only ir device
 90 struct irda_device *ir_dev;
 91
 92 // queue of sk_buffs to transmit
 93 struct sk_buff_head tx_queue;
 94
 95 // queue taking received frames
 96 struct sk_buff_head rx_queue;
 97 int rx_queue_max_len;
 98
 99 // buffers to store received bytes and
100 // bytes to be unwrapped
101 #define IR_REC_BUF_SIZE 2200
102 volatile static int ir_rec_count = 0;
103 static UC8 *ir_rec_buf;
104 static UC8 *ir_unwrap_buf;
105
106
107 // code to get around the SA1100 SIR-bug
108 #ifdef SA1100
109     RingBuffer irport_tx_buf;
110     int irport_ir_tx_buf_size=IRPORT_TRANSM_BUFSIZE; //SIR transmit buffer size
111
112     // timer channel to be used by the SIR transmission
113     int irport_sir_ostimer_channel =  OSTIMER_CHANNEL_3;
114
```

Listing 54. Listing of irport.c

```
115     int irport_sir_ostimer_int; // interrupt for the SIR transmission
116     BOOL irport_sir_transmitting = FALSE; // transmitting
117
118     static volatile unsigned char ir_ch; // current char being transmitted
119     static volatile unsigned int ir_state = FIRST_PERIOD;  // transm. state
120     static volatile int bit_count=10;
121
122     int irport_sir_init(int ostimer_channel);
123     #define IRPORT_SIR_INIT  irport_sir_init(irport_sir_ostimer_channel)
124
125 #else
126     #define TRANSM_BUSY_DELAY 300  // allow UART TBY flag to be set
127     int irport_sir_init(void);
128     #define IRPORT_SIR_INIT  irport_sir_init()
129     __inline void irport_write_wakeup(void);
130 #endif
131
132 // toggle between receiver and transmitter mode
133 __inline void irport_sir_set_receive_mode(void);
134 __inline void irport_sir_set_transmit_mode(void);
135
136
137
138 static int  irport_net_open(void);
139 static int  irport_net_close(void);
140 static int  irport_raw_write(__u8 *buf, int len);
141
142
143
144 int irport_open(void)
145 {
146     DEBUG(4, “irport_open()\n”);
147     /*
148      *  Allocate new instance of the driver
149      */
150     ir_dev = (struct irda_device*)malloc(sizeof(struct irda_device));
151     if (ir_dev == NULL) {
152         ERROR(“IrDA-irport_open: Can’t allocate memory for “
153             “IrDA control block!\n”);
154         return -ENOMEM;
155     }
156     memset(ir_dev, 0, sizeof(struct irda_device));
157
158     /* Initialize QoS for this device */
159     irda_init_max_qos_capabilies(&ir_dev->qos);
160
161     // activate the bottom half service
162     ostimer_SetBHActive();
163
164 #ifdef SA1100   // only software modulation with 9600, FIR not yet implemented
165     ir_dev->qos.baud_rate.bits = IR_9600; //|IR_4000000;
166 #else // all modes, but FIR not yet implemented
167     ir_dev->qos.baud_rate.bits = IR_9600|IR_19200|IR_38400|IR_57600|
168         IR_115200; //| IR_4000000;
169 #endif
170
171
172     // init the receiver and transmitter queues and buffers
173     // and the minimum link turnaround time
174     ir_dev->qos.min_turn_time.bits = qos_mtt_bits;
175     irda_qos_bits_to_value(&ir_dev->qos);
176
177     skb_queue_head_init(&tx_queue);
178     skb_queue_head_init(&rx_queue);
179
```

Listing 54. Listing of irport.c

```
180     ir_dev->rx_buff.truesize = 4000;
181     ir_dev->tx_buff.truesize = 4000;
182
183
184     /*
185      *  Allocate receiver interrupt buffer
186      */
187     ir_rec_buf = (UC8*)malloc(IR_REC_BUF_SIZE);
188     if (ir_rec_buf == NULL) {
189         ERROR("IrDA-irport_open: Can't allocate memory for "
190             "receiver interrupt buffer!\n");
191         return -ENOMEM;
192     }
193
194     /*
195      *  Allocate unwrap buffer
196      */
197     ir_unwrap_buf = (UC8*)malloc(IR_REC_BUF_SIZE);
198     if (ir_unwrap_buf == NULL) {
199         ERROR("IrDA-irport_open: Can't allocate memory for "
200             "unwrap buffer!\n");
201         return -ENOMEM;
202     }
203
204     /* Open the IrDA device */
205     irda_device_open();
206
207     irda_device_setup();
208     irport_net_open();
209
210     return 0;
211 }
212
213 int irport_close(void)
214 {
215     ASSERT(ir_dev != NULL, return -1;);
216
217     ostimer_SetBHInactive();
218
219     irda_device_close();
220
221     free(ir_dev);
222
223     return 0;
224 }
225
226
227 /*
228  * Function irport_net_open (dev)
229  */
230 static int irport_net_open(void)
231 {
232     // init the irport, install the interrupt handlers,...
233     IRPORT_SIR_INIT;
234
235
236     /* Ready to play! */
237     ir_dev->tbusy = 0;
238     ir_dev->start = 1;
239
240     return 0;
241 }
242
243
244 /*
```

Listing 54. Listing of irport.c

```
245  * Function irport_net_close (ir_dev)
246  */
247 static int irport_net_close(void)
248 {
249     /* Stop device */
250     ir_dev->tbusy = 1;
251     ir_dev->start = 0;
252 
253     irport_stop();
254 
255     return 0;
256 }
257 
258 
259 void irport_start(void)
260 {
261     unsigned long flags;
262 
263     spin_lock_irqsave(&ir_dev->lock, flags);
264 
265     irport_stop();
266 
267     /* Initialize UART */
268 
269 //  /* Turn on interrups */
270     IR_HSSP_ENABLE_TRI;
271 #ifdef SA1100   // for old SA1100 only receiver interupt,
272                 // transmission by software modulation
273     SER_UART_ENABLE_RI(IR_UART_BASE);
274 #else
275     SER_UART_ENABLE_TRI(IR_UART_BASE);
276 #endif
277 
278     spin_unlock_irqrestore(&ir_dev->lock, flags);
279 }
280 
281 void irport_stop(void)
282 {
283     unsigned long flags;
284 
285     spin_lock_irqsave(&ir_dev->lock, flags);
286 
287     /* shutdown serial port 2 */
288     IR_HSSP_SHUTDOWN;
289     SER_UART_SHUTDOWN(IR_UART_BASE);
290 
291     spin_unlock_irqrestore(&ir_dev->lock, flags);
292 }
293 
294 void irport_sir_int_handler(unsigned int ident, unsigned int data,
295                             unsigned int empty_stack);
296 
297 /*
298  * Function irport_change_speed (ir_dev, speed)
299  *
300  *    Set speed of IrDA port to specified baudrate
301  *
302  */
303 void irport_change_speed(int speed)
304 {
305     unsigned long flags;
306 
307     ASSERT(ir_dev != NULL, return;);
308 
309     /* Update accounting for new speed */
```

Listing 54. Listing of irport.c

```
310
311     spin_lock_irqsave(&ir_dev->lock, flags);
312
313     if(speed < 4000000)
314     {
315
316         DEBUG_1(4, “irport_change_speed SIR, %d baud\n”, speed);
317         // turn off all FIR related functions
318         REG(HSSP_BASE, HSCR0) = 0;
319
320         // IR-transceiver mode
321         IR_SEL_SIR; // select sir mode for SA1100 and transceiver
322
323 #ifdef SA1100
324         if(speed == 9600)
325         {
326             PPC_SET_OUTPUT(TXD2);   // set txd2 (SIR transmit) to output
327             PPC_CLEAR_PIN(TXD2);    // and clear it
328         }
329         else
330         {
331             ERROR(“only 9600 baud supported with SA1100 SIR\n”);
332         }
333 #else
334         REG(IR_UART_BASE, UTCR3) = 0x00000000;      // shutdown UART
335         SER_UART_CLEAR_STATUS_BITS_ALL(IR_UART_BASE);   // clear any status bits
336         switch(speed)
337         {
338             case 19200:     REG(IR_UART_BASE, UTCR2) = 0x0000000B;
339                             break;
340
341             case 38400:     REG(IR_UART_BASE, UTCR2) = 0x00000005;
342                             break;
343
344             case 57600:     REG(IR_UART_BASE, UTCR2) = 0x00000003;
345                             break;
346
347             case 115200:    REG(IR_UART_BASE, UTCR2) = 0x00000001;
348                             break;
349
350             // 9600 baud as default
351             default:        REG(IR_UART_BASE, UTCR2) = 0x00000017;
352                             break;
353         }
354
355 #endif  // SA1100
356
357         int_InstallHandler(IR_UART_INT, irport_sir_int_handler);
358
359     }
360     else
361     {
362         DEBUG(4, “irport_change_speed FIR\n”);
363         REG(IR_UART_BASE, UTCR3) = 0x00000000;      // shutdown UART
364         IR_SEL_FIR;
365         IR_HSSP_EN(1,1);
366
367 //      FIR mode is not yet implemented
368 //      int_InstallHandler(IR_UART_INT, irport_fir_int_handler);
369 //        DEBUG_STRING(“FIR not yet implemented\n\r”);
370         printf(“FIR not yet implemented, exit\n”);
371         exit(1);
372     }
373     /* Turn on interrups */
374
```

Listing 54. Listing of irport.c

```
375     ir_dev->io.baudrate = speed;
376
377     spin_unlock_irqrestore(&ir_dev->lock, flags);
378 }
379
380
381 // set sir to receive mode
382 __inline void irport_sir_set_receive_mode(void)
383 {
384     UC8 dummy;
385
386 #ifndef SA1100
387     {
388         int delay;
389
390         // wait until transmission is finished. Not necessary if the
391         // software modulation is used
392
393         // UART TBY flag not immediately set after transmitter enable.
394         // We poll the TBY flag to see when we are finished with the
395         // transmission and can switch to receiver mode. If bit is
396         // queried immediately after starting the transmission it might
397         // not yet be set and thus we immediately switch to receiver
398         // mode and loose complete transmission.  SA1100 bug ?
399         // Shouldn't be a problem here as some code should have been
400         // executed in between, but just to make sure
401         for(delay=0;delay<TRANSM_BUSY_DELAY;delay++);
402     }
403     while (SER_UART_TRANSM_BUSY_Q(IR_UART_BASE))
404     {
405         ;
406     }
407
408     // disable transmitter and transmitter interrupt
409     SER_UART_DISABLE_T(IR_UART_BASE);
410     SER_UART_DISABLE_TI(IR_UART_BASE);
411 #endif
412
413     // make sure receiver FIFO doesn't contain any reflected data
414     while(SER_UART_REC_NOT_EMPTY_Q(IR_UART_BASE))
415     {
416         dummy = REG(IR_UART_BASE, UTDR);
417     }
418
419     // clear status bits and then enable receiver interrupt and receiver
420     SER_UART_CLEAR_STATUS_BITS_ALL(IR_UART_BASE);
421     SER_UART_ENABLE_RI(IR_UART_BASE);
422     SER_UART_ENABLE_R(IR_UART_BASE);
423
424 }
425
426
427 // set sir to transmit mode
428 __inline void irport_sir_set_transmit_mode(void)
429 {
430     // read any bytes left in the receiver FIFO
431     // should cause int but for some reason doesn't do always
432     while(SER_UART_REC_NOT_EMPTY_Q(IR_UART_BASE))
433     {
434         async_unwrap_char(ir_dev, REG(IR_UART_BASE, UTDR));
435     }
436
437     // disable receiver and receiver interrupt
438     SER_UART_DISABLE_R(IR_UART_BASE);
439     SER_UART_DISABLE_RI(IR_UART_BASE);
```

Listing 54. Listing of irport.c

```
440
441
442     // clear status bits and then enable transmit interrupt and transmitter
443
444     SER_UART_CLEAR_STATUS_BITS_ALL(IR_UART_BASE);
445 #ifndef SA1100
446     SER_UART_ENABLE_TI(IR_UART_BASE);
447 #endif
448     SER_UART_ENABLE_T(IR_UART_BASE);
449 }
450
451
452
453 #ifdef SA1100
454
455
456 extern volatile int ostimer_blocked;
457 extern int irlap_last_discovery;
458
459
460 // start transmission via software modulation
461 void irport_sir_transmit_data(void)
462 {
463     // start transmission if not already transmitting
464     if(!irport_sir_transmitting)
465     {
466         // block software timer callbacks to ensure proper transmission
467         ostimer_blocked = TRUE;
468         irport_sir_transmitting = TRUE;
469         DEBUG_STRING("irport start transm.\n\r");
470
471
472         OSTIMER_INC_MATCH_REG(irport_sir_ostimer_channel, 40);
473         INT_UNMASK(irport_sir_ostimer_int);
474
475     }
476     else
477     {
478         DEBUG_STRING("irport transm. busy\n\r");
479     }
480 }
481
482
483
484 void ir_SirTransmitIntHandler(unsigned int ident, unsigned int data,
485                               unsigned int empty_stack)
486 {
487
488     // last byte finished, get new byte
489     if(bit_count>=10 && ir_dev->tx_buff.len>0)
490     {
491         ir_ch = *(ir_dev->tx_buff.data++);
492         ir_dev->tx_buff.len--;
493         bit_count=0;
494     }
495
496     if(ir_state==FIRST_PERIOD)
497     {
498         if(bit_count==0 ||
499             (!(ir_ch & 1<<(bit_count-1)) && bit_count <= 8)   )
500         {
501             // for 0-bit 3/16 high period
502             OSTIMER_INC_MATCH_REG(irport_sir_ostimer_channel,IR_OSM_FACTOR*3);
503
504             PPC_SET_PIN(TXD2);
```

Listing 54. Listing of irport.c

```
505
506             ir_state = SECOND_PERIOD;   // we need a 13/16 low period
507
508         }
509         else    // data bit is 1 -> zero period or it is the stop bit
510         {
511             // 16/16 low period
512             OSTIMER_INC_MATCH_REG(irport_sir_ostimer_channel,
513                                   IR_OSM_FACTOR*16);
514             bit_count++;
515         }
516     }
517     else
518     {
519         // 13/16 low period as second half
520         OSTIMER_INC_MATCH_REG(irport_sir_ostimer_channel, IR_OSM_FACTOR*13);
521
522         PPC_CLEAR_PIN(TXD2);
523         ir_state = FIRST_PERIOD;
524         bit_count++;
525     }
526     OSTIMER_RESET_INT(irport_sir_ostimer_channel);
527
528     // byte finished and transfer of buffer finished, disable interrupt
529     if(bit_count>=10 && ir_dev->tx_buff.len==0)
530     {
531
532         INT_MASK(irport_sir_ostimer_int);
533         irport_sir_set_receive_mode();
534         irport_sir_transmitting=FALSE;
535         INT_UNMASK(INT_OSTIMER_0+1);
536
537         // get new frame or shut down
538         ostimer_MarkBH(OSTIMER_BH_IRPORT_TRANSMIT);
539
540         ir_dev->tbusy = 0; // Unlock
541         ostimer_blocked = FALSE;    // unblock software timer callbacks
542         DEBUG_STRING("irport unlocked\n\r");
543     }
544 }
545
546 #else
547
548 extern volatile int ostimer_blocked;
549
550 /*
551  * Function irport_write_wakeup (tty)
552  *
553  *    Called by the interrupt when there's room for more data.  If we have
554  *    more packets to send, we send them here.
555  *
556  */
557 __inline void irport_write_wakeup(void)
558 {
559     int i;
560     int actual = 0;
561
562     ASSERT(ir_dev != NULL, return;);
563
564     // Finished with frame?
565     if (ir_dev->tx_buff.len > 0)  {
566
567         // Write data left in transmit buffer, if transmit request int
568         // four bytes can be written to the fifo without check
569         for(i=0;i<4 && actual < ir_dev->tx_buff.len; i++)
```

Listing 54. Listing of irport.c

```
570         {
571             REG(IR_UART_BASE, UTDR) = ir_dev->tx_buff.data[actual++];
572         }
573 
574         // if transmitter not yet full fill it up
575         while( TEST_BIT(IR_UART_BASE, UTSR1, UTSR1_TNF) &&
576                actual < ir_dev->tx_buff.len )
577         {
578             REG(IR_UART_BASE, UTDR) = ir_dev->tx_buff.data[actual++];
579         }
580 
581         ir_dev->tx_buff.data += actual;
582         ir_dev->tx_buff.len  -= actual;
583 
584 
585     // buffer empty
586     } else {
587 
588         char deb_buf[50];
589 
590         irport_sir_set_receive_mode();
591 
592         // Schedule network layer, so we can get some more frames
593         ostimer_MarkBH(OSTIMER_BH_IRPORT_TRANSMIT);
594 
595         // Now serial buffer is almost free & we can start
596         // transmission of another packet
597         ir_dev->tbusy = 0; // Unlock
598 
599         ostimer_blocked = FALSE;
600     }
601 }
602 
603 #endif  // #ifdef SA1100
604 
605 
606 /*
607  * Function irport_hard_xmit (void)
608  *
609  *    Transmits the current frame until FIFO is full, then
610  *    waits until the next transmitt interrupt, and continues until the
611  *    frame is transmited.
612  */
613 int irport_hard_xmit(struct sk_buff *skb)
614 {
615     unsigned long flags;
616 
617     ASSERT(ir_dev != NULL, return 0;);
618 
619     spin_lock_irqsave(&ir_dev->lock, flags);
620 
621     ir_dev->tbusy = TRUE;
622 
623     /* Init tx buffer */
624     ir_dev->tx_buff.data = ir_dev->tx_buff.head;
625 
626     /* Copy skb to tx_buff while wrapping, stuffing and making CRC */
627     ir_dev->tx_buff.len = async_wrap_skb(skb, ir_dev->tx_buff.data,
628                         ir_dev->tx_buff.truesize);
629 
630 /* Turn on transmit interrupt to start transmission. */
631     irport_sir_set_transmit_mode();
632 
633 #ifdef SA1100
634     irport_sir_transmit_data();
```

Listing 54. Listing of irport.c

```
635 #endif
636
637     spin_unlock_irqrestore(&ir_dev->lock, flags);
638
639     kfree_skb(skb);
640
641     return 0;
642 }
643
644
645
646
647 /*
648  * Function irport_interrupt (irq, dev_id, regs)
649  *
650  *    Interrupt handler for SIR mode
651  */
652
653 void irport_sir_int_handler(unsigned int ident, unsigned int data,
654                             unsigned int empty_stack)
655 {
656     char ch;
657     UC8 utsr0;
658     int i;
659
660
661
662     if(TEST_BIT(IR_UART_BASE, UTSR0, UTSR0_RFS))    // receiver fifo request
663
664     {
665         // put received bytes into irport receiver buffer, can be unstuffed
666         // later in the bottom half handler
667         ir_rec_buf[ir_rec_count++] = REG(IR_UART_BASE, UTDR);
668         ir_rec_buf[ir_rec_count++] = REG(IR_UART_BASE, UTDR);
669         ir_rec_buf[ir_rec_count++] = REG(IR_UART_BASE, UTDR);
670         ir_rec_buf[ir_rec_count++] = REG(IR_UART_BASE, UTDR);
671
672         while(SER_UART_REC_NOT_EMPTY_Q(IR_UART_BASE))
673         {
674             ir_rec_buf[ir_rec_count++] = REG(IR_UART_BASE, UTDR);
675         }
676         ostimer_MarkBH(OSTIMER_BH_IRPORT_UNWRAP);
677
678     }
679
680     else if(TEST_BIT(IR_UART_BASE, UTSR0, UTSR0_RID))   // receiver idle
681     {
682         do
683         {
684             ir_rec_buf[ir_rec_count++] = REG(IR_UART_BASE, UTDR);
685         }
686         while(SER_UART_REC_NOT_EMPTY_Q(IR_UART_BASE));
687         ostimer_MarkBH(OSTIMER_BH_IRPORT_UNWRAP);
688         SER_UART_CLEAR_STATUS_BITS_RID(IR_UART_BASE);
689     }
690
691     // transmit interrupt request
692     if(TEST_BIT(IR_UART_BASE, UTSR0, UTSR0_TFS))
693     {
694 #ifdef SA1100
695         INT_MASK(IR_UART_INT);
696 #else
697     irport_write_wakeup();
698 #endif
699     }
```

Listing 54. Listing of irport.c

```
700
701     if(TEST_BIT(IR_UART_BASE, UTSR0, UTSR0_RBB))    // receiver break begin
702     {
703         SER_UART_CLEAR_STATUS_BITS_RBB(IR_UART_BASE);
704     }
705
706     if(TEST_BIT(IR_UART_BASE, UTSR0, UTSR0_REB))    // receiver break end
707     {
708         SER_UART_CLEAR_STATUS_BITS_REB(IR_UART_BASE);
709     }
710
711     // don't care, will be detected in the CRC check
712     if(TEST_BIT(IR_UART_BASE, UTSR0, UTSR0_EIF))    // error in frame
713     {
714         do
715         {
716             ir_rec_buf[ir_rec_count++] = REG(IR_UART_BASE, UTDR);
717         }
718         while(TEST_BIT(IR_UART_BASE, UTSR0, UTSR0_EIF));
719         ostimer_MarkBH(OSTIMER_BH_IRPORT_UNWRAP);
720     }
721
722
723     utsr0 = REG(IR_UART_BASE, UTSR0);   // find reason for int request
724     if(utsr0 )                          // shouldn't be set anymore, otherwise error
725     {
726         INT_MASK(IR_UART_INT);
727     }
728
729 }
730
731
732 /* initialize uart2 as infrared port in serial mode
733  */
734 #ifdef SA1100
735     int irport_sir_init(int ostimer_channel)
736 #else
737     int irport_sir_init(void)
738 #endif
739
740 {
741
742     DEBUG(4, "irport_sir_init()\n");
743
744     REG(HSSP_BASE, HSCR0) = 0;
745
746     REG(IR_UART_BASE, UTCR3) = 0x00000000;      // shutdown UART
747     SER_UART_CLEAR_STATUS_BITS_ALL(IR_UART_BASE);   // clear any status bits
748     REG(IR_UART_BASE, UTCR0) = 0x00000008;      // 8N1
749     REG(IR_UART_BASE, UTCR1) = 0x00000000;      // 9600 baud
750     REG(IR_UART_BASE, UTCR2) = 0x00000017;      // 9600 baud
751
752     IR_UART_SIR_ENABLE;   // activate SIR modulation for UART2
753
754     // IR-transceiver mode
755     IR_SEL_SIR;     // select sir mode for SA1100 and transceiver
756
757 #ifdef SA1100
758     irport_tx_buf.data = (unsigned char*) malloc(IRPORT_TRANSM_BUFSIZE);
759     if(irport_tx_buf.data)
760     {
761         irport_tx_buf.size = IRPORT_TRANSM_BUFSIZE;
762     }
763     else
764     {
```

Listing 54. Listing of irport.c

```
765         printf("not enough memory for ir transmit buffer\n");
766         exit(1);
767     }
768
769     PPC_SET_OUTPUT(TXD2);   // set txd2 (SIR transmit) to output
770     PPC_CLEAR_PIN(TXD2);    // and clear it
771
772     irport_sir_ostimer_channel = ostimer_channel;
773     OSTIMER_ENABLE_INT(irport_sir_ostimer_channel);
774     irport_sir_ostimer_int = INT_OSTIMER_0+irport_sir_ostimer_channel;
775     int_InstallHandler(irport_sir_ostimer_int, ir_SirTransmitIntHandler);
776
777     // sofware modulation doesn't work otherwise
778     misc_SysDisableWriteBuffer();
779
780 #endif
781
782     int_InstallHandler(IR_UART_INT, irport_sir_int_handler);
783     INT_UNMASK(IR_UART_INT);
784
785     return 0;
786 }
787
788
789
790
791 /*
792  * Function irport_wait_until_sent (ir_dev)
793  *
794  *    Delay execution until finished transmitting
795  *
796  */
797 void irport_wait_until_sent(void)
798 {
799
800 // FIXME maybe something like yield
801
802 #ifdef SA1100
803     while (irport_sir_transmitting)
804         ;
805 #else
806     {
807         int delay;
808
809         // UART TBY flag not immediately set after transmitter enable.
810         // Shouldn't be a problem here as some code should have been
811         // executed in between, but just to make sure
812         for(delay=0;delay<TRANSM_BUSY_DELAY;delay++);
813     }
814     /* Wait until Tx FIFO is empty */
815     while (SER_UART_TRANSM_BUSY_Q(IR_UART_BASE))
816     {
817         ;
818
819     }
820
821 #endif
822
823 }
824
825 /*
826  * Function irport_is_receiving (ir_dev)
827  *
828  *    Returns true if we are currently receiving data
829  *
```

Listing 54. Listing of irport.c

```
830  */
831 int irport_is_receiving(void)
832 {
833     return (ir_dev->rx_buff.state != OUTSIDE_FRAME);
834 }
835
836
837
838 /*
839  * bottom half handler for transmission
840  * if device (i.e. port) is not busy and transmission queue is not
841  * empty take out a buffer and pass it to irport_hard_xmit for
842  * transmission
843  */
844 void irport_BHTransmit(void)
845 {
846     struct sk_buff* tx_skb;
847
848     // reset the request
849     ostimer_UnmarkBH(OSTIMER_BH_IRPORT_TRANSMIT);
850
851     // if there is a packet to transmit
852     if(tx_queue.qlen>0)
853     {
854
855         if(!ir_dev->tbusy)
856         {
857             // get the sk_buff and pass it to the transmission routine
858             tx_skb = skb_dequeue(&tx_queue);
859             irport_hard_xmit(tx_skb);
860         }
861         else
862         {
863             // when transmission is finished the int handler marks
864             // the bottom half handler again for the next packet
865         }
866     }
867
868 }
869
870
871 /*
872  * Bottom half receive function.
873  * Take queued skbs out of the receiver queue and pass them to the
874  * irlap receive function.
875  */
876 void irport_BHReceive(void)
877 {
878     struct sk_buff* rx_skb;
879
880     ostimer_UnmarkBH(OSTIMER_BH_IRPORT_RECEIVE);
881
882     while(rx_queue.qlen>0)
883     {
884         rx_skb = skb_dequeue(&rx_queue);
885         irlap_driver_rcv(rx_skb);
886     }
887
888
889 }
890
891
892 /*
893  * Bottom half unwrap function.
894  * copy received bytes from ir_rec_buf to ir_unwrap_buf to avoid
```

Listing 54. Listing of irport.c

```
 * blocking interrupts for a longer time. Unwrapping then can be
 * done in user mode while the interrupt can receive the next
 * bytes.
 */
void irport_BHUnwrap(void)
{
    int i;
    int len;
    ostimer_UnmarkBH(OSTIMER_BH_IRPORT_UNWRAP);

    Angel_EnterSVC();
    memcpy(ir_unwrap_buf, ir_rec_buf, ir_rec_count);
    len = ir_rec_count;
    ir_rec_count=0;
    Angel_ExitToUSR();

    for(i=0;i<len;i++)
    {
        async_unwrap_char(ir_dev, ir_unwrap_buf[i]);
    }

}
```

Listing 54. Listing of irport.c

C.2 Implementation of wrapper.h

```
/*********************************************************************
 *
 * Filename:      wrapper.h
 * Version:       1.2
 * Description:   IrDA SIR async wrapper layer
 * Status:        Experimental.
 * Author:        Dag Brattli <dagb@cs.uit.no>
 * Created at:    Mon Aug  4 20:40:53 1997
 * Modified at:   Mon May  3 09:02:36 1999
 * Modified by:   Dag Brattli <dagb@cs.uit.no>
 *
 *     Copyright (c) 1998-1999 Dag Brattli <dagb@cs.uit.no>,
 *     All Rights Reserved.
 *
 *     This program is free software; you can redistribute it and/or
 *     modify it under the terms of the GNU General Public License as
 *     published by the Free Software Foundation; either version 2 of
 *     the License, or (at your option) any later version.
 *
 *     Neither Dag Brattli nor University of Tromsø admit liability nor
 *     provide warranty for any of this software. This material is
 *     provided “AS-IS” and at no charge.
 *
 *
 * Modified by: Christoph Wolf <chwolf@it.kth.se> for the
 *                 SmartBadge IrDA-Project, Nov. 2000
 *
 *
 ********************************************************************/

#ifndef WRAPPER_H
#define WRAPPER_H

#include <irda/types.h>
```

Listing 55. Listing of wrapper.h

```
35 #include <irda/skbuff.h>
36
37 #include <irda/irda_device.h>
38
39 #define BOF  0xc0 /* Beginning of frame */
40 #define XBOF 0xff
41 #define IRDA_EOF  0xc1 /* End of frame */
42 #define CE   0x7d /* Control escape */
43
44 #define STA BOF  /* Start flag */
45 #define STO IRDA_EOF  /* End flag */
46
47 #define IRDA_TRANS 0x20    /* Asynchronous transparency modifier */
48
49 /* States for receving a frame in async mode */
50 enum {
51     OUTSIDE_FRAME,
52     BEGIN_FRAME,
53     LINK_ESCAPE,
54     INSIDE_FRAME
55 };
56
57 /* Proto definitions */
58 int async_wrap_skb(struct sk_buff *skb, __u8 *tx_buff, int buffsize);
59
60 /*
61  * Function async_unwrap (skb)
62  *
63  *    Parse and de-stuff frame received from the IrDA-port
64  *
65  */
66
67 extern void (*async_wrap_state[])(struct irda_device *idev, __u8 byte);
68 inline void async_unwrap_char(struct irda_device *idev, __u8 byte)
69 {
70     (*async_wrap_state[idev->rx_buff.state]) (idev, byte);
71 }
72
73 #endif
```

Listing 55. Listing of wrapper.h

C.3 Implementation of wrapper.c

```
 1 /*********************************************************************
 2  *
 3  * Filename:      wrapper.c
 4  * Version:       1.2
 5  * Description:   IrDA SIR async wrapper layer
 6  * Status:        Experimental.
 7  * Author:        Dag Brattli <dagb@cs.uit.no>
 8  * Created at:    Mon Aug  4 20:40:53 1997
 9  * Modified at:   Fri May 28 20:30:24 1999
10  * Modified by:   Dag Brattli <dagb@cs.uit.no>
11  *
12  *     Copyright (c) 1998-1999 Dag Brattli <dagb@cs.uit.no>,
13  *     All Rights Reserved.
14  *
15  *     This program is free software; you can redistribute it and/or
16  *     modify it under the terms of the GNU General Public License as
17  *     published by the Free Software Foundation; either version 2 of
```

Listing 56. Listing of wrapper.c

```
 *     the License, or (at your option) any later version.
 *
 *     Neither Dag Brattli nor University of Tromsø admit liability nor
 *     provide warranty for any of this software. This material is
 *     provided “AS-IS” and at no charge.
 *
 *
 * Modified by: Christoph Wolf <chwolf@it.kth.se> for the
 *                 SmartBadge IrDA-Project, Nov. 2000
 *
 *
 ********************************************************************/

#include <irda/skbuff.h>
#include <string.h>
#include <irda/irda.h>
#include <irda/crc.h>
#include <irda/irlap.h>
#include <irda/irlap_frame.h>
#include <irda/irda_device.h>
#include <irda/netdevice.h>
#include <irda/wrapper.h>

#include <util/util_ostimer.h>
#include <util/util_debug.h>



static inline int stuff_byte(__u8 byte, __u8 *buf);

static void state_outside_frame(struct irda_device *idev, __u8 byte);
static void state_begin_frame(struct irda_device *idev, __u8 byte);
static void state_link_escape(struct irda_device *idev, __u8 byte);
static void state_inside_frame(struct irda_device *idev, __u8 byte);

extern struct sk_buff_head rx_queue;
extern int rx_queue_max_len;
extern int rx_queue_new_data;

void (*async_wrap_state[])(struct irda_device *idev, __u8 byte) =
{
    state_outside_frame,
    state_begin_frame,
    state_link_escape,
    state_inside_frame,
};

/*
 * Function async_wrap (skb, *tx_buff)
 *
 *    Makes a new buffer with wrapping and stuffing, should check that
 *    we don’t get tx buffer overflow.
 */
int async_wrap_skb(struct sk_buff *skb, __u8 *tx_buff, int buffsize)
{
    int i;
    int n;
    int xbofs;
    union {
        __u16 value;
        __u8 bytes[2];
    } fcs;

    /* Initialize variables */
    fcs.value = INIT_FCS;
```

Listing 56. Listing of wrapper.c

```
 83     n = 0;
 84
 85     if (skb->len > 2048) {
 86         DEBUG_1(0, "async_wrap_skb(): Warning size=%d of sk_buff to big!\n",
 87               (int) skb->len);
 88         return 0;
 89     }
 90
 91     /*
 92      *  Send  XBOF's for required min. turn time and for the negotiated
 93      *  additional XBOFS
 94      */
 95     xbofs = ((struct irlap_skb_cb *)(skb->cb))->xbofs;
 96
 97     memset(tx_buff+n, XBOF, xbofs);
 98     n += xbofs;
 99
100     /* Start of packet character BOF */
101     tx_buff[n++] = BOF;
102
103     /* Insert frame and calc CRC */
104     for (i=0; i < skb->len; i++) {
105         /*
106          *  Check for the possibility of tx buffer overflow. We use
107          *  bufsize-5 since the maximum number of bytes that can be
108          *  transmitted after this point is 5.
109          */
110         ASSERT(n < (buffsize-5), return n;);
111
112         n += stuff_byte(skb->data[i], tx_buff+n);
113         fcs.value = irda_fcs(fcs.value, skb->data[i]);
114     }
115
116     /* Insert CRC in little endian format (LSB first) */
117     fcs.value = ~fcs.value;
118 #ifdef __LITTLE_ENDIAN
119     n += stuff_byte(fcs.bytes[0], tx_buff+n);
120     n += stuff_byte(fcs.bytes[1], tx_buff+n);
121 #else
122     #ifdef __BIG_ENDIAN
123         n += stuff_byte(fcs.bytes[1], tx_buff+n);
124         n += stuff_byte(fcs.bytes[0], tx_buff+n);
125     #else
126             #error neither __LITTLE_ENDIAN nor __BIG_ENDIAN defined
127     #endif
128 #endif
129     tx_buff[n++] = IRDA_EOF;
130
131     return n;
132 }
133
134 /*
135  * Function async_bump (idev)
136  *
137  *    Got a frame, make a copy of it, and pass it up the stack!
138  *
139  */
140 static inline void async_bump(struct irda_device *idev, __u8 *buf, int len)
141 {
142     struct sk_buff *skb;
143
144     skb = dev_alloc_skb(len+1);
145     if (!skb)  {
146         return;
147     }
```

Listing 56. Listing of wrapper.c

```
148
149     /*  Align IP header to 20 bytes */
150     skb_reserve(skb, 1);
151
152         /* Copy data without CRC */
153     memcpy(skb_put(skb, len-2), buf, len-2);
154
155     /*
156      *  Feed it to IrLAP layer
157      */
158
159     if(rx_queue.qlen <= rx_queue_max_len)
160     {
161         skb_queue_tail(&rx_queue, skb);
162         ostimer_MarkBH(OSTIMER_BH_IRPORT_RECEIVE);
163     }
164 }
165
166 /*
167  * Function stuff_byte (byte, buf)
168  *
169  *    Byte stuff one single byte and put the result in buffer pointed to by
170  *    buf. The buffer must at all times be able to have two bytes inserted.
171  *
172  */
173 static inline int stuff_byte(__u8 byte, __u8 *buf)
174 {
175     switch (byte) {
176     case BOF: /* FALLTHROUGH */
177     case IRDA_EOF: /* FALLTHROUGH */
178     case CE:
179         /* Insert transparently coded */
180         buf[0] = CE;               /* Send link escape */
181         buf[1] = byte^IRDA_TRANS;    /* Complement bit 5 */
182         return 2;
183         /* break; */
184     default:
185          /* Non-special value, no transparency required */
186         buf[0] = byte;
187         return 1;
188         /* break; */
189     }
190 }
191
192
193 /*
194  * Function state_outside_frame (idev, byte)
195  *
196  *
197  *
198  */
199 static void state_outside_frame(struct irda_device *idev, __u8 byte)
200 {
201     switch (byte) {
202     case BOF:
203         idev->rx_buff.state = BEGIN_FRAME;
204         idev->rx_buff.in_frame = TRUE;
205         break;
206     case XBOF:
207         /* idev->xbofs++; */
208         break;
209     case IRDA_EOF:
210         irda_device_set_media_busy(TRUE);
211         break;
212     default:
```

Listing 56. Listing of wrapper.c

```
213         break;
214     }
215 }
216
217 /*
218  * Function state_begin_frame (idev, byte)
219  *
220  *    Begin of frame detected
221  *
222  */
223 static void state_begin_frame(struct irda_device *idev, __u8 byte)
224 {
225     switch (byte) {
226     case BOF:
227         /* Continue */
228         break;
229     case CE:
230         /* Stuffed byte */
231         idev->rx_buff.state = LINK_ESCAPE;
232
233         /* Time to initialize receive buffer */
234         idev->rx_buff.data = idev->rx_buff.head;
235         idev->rx_buff.len = 0;
236         break;
237     case IRDA_EOF:
238         /* Abort frame */
239         idev->rx_buff.state = OUTSIDE_FRAME;
240         break;
241     default:
242         /* Time to initialize receive buffer */
243         idev->rx_buff.data = idev->rx_buff.head;
244         idev->rx_buff.len = 0;
245
246         idev->rx_buff.data[idev->rx_buff.len++] = byte;
247
248         idev->rx_buff.fcs = irda_fcs(INIT_FCS, byte);
249         idev->rx_buff.state = INSIDE_FRAME;
250         break;
251     }
252 }
253
254 /*
255  * Function state_link_escape (idev, byte)
256  *
257  *
258  *
259  */
260 static void state_link_escape(struct irda_device *idev, __u8 byte)
261 {
262     switch (byte) {
263     case BOF: /* New frame? */
264         idev->rx_buff.state = BEGIN_FRAME;
265         irda_device_set_media_busy(TRUE);
266         break;
267     case CE:
268         DEBUG(4, "WARNING: state_link_escape() State not defined\n");
269         break;
270     case IRDA_EOF: /* Abort frame */
271         idev->rx_buff.state = OUTSIDE_FRAME;
272         break;
273     default:
274         /*
275          *  Stuffed char, complement bit 5 of byte
276          *  following CE, IrLAP p.114
277          */
```

Listing 56. Listing of wrapper.c

```
278         byte ^= IRDA_TRANS;
279         if (idev->rx_buff.len < idev->rx_buff.truesize)  {
280             idev->rx_buff.data[idev->rx_buff.len++] = byte;
281             idev->rx_buff.fcs = irda_fcs(idev->rx_buff.fcs, byte);
282             idev->rx_buff.state = INSIDE_FRAME;
283         } else {
284             DEBUG(1, “state_link_escape(), Rx buffer overflow, aborting\n”);
285             idev->rx_buff.state = OUTSIDE_FRAME;
286         }
287         break;
288     }
289 }
290
291 /*
292  * Function state_inside_frame (idev, byte)
293  *
294  *    Handle bytes received within a frame
295  *
296  */
297 static void state_inside_frame(struct irda_device *idev, __u8 byte)
298 {
299
300     switch (byte) {
301     case BOF: /* New frame? */
302         idev->rx_buff.state = BEGIN_FRAME;
303         irda_device_set_media_busy(TRUE);
304         break;
305     case CE: /* Stuffed char */
306         idev->rx_buff.state = LINK_ESCAPE;
307         break;
308     case IRDA_EOF: /* End of frame */
309         idev->rx_buff.state = OUTSIDE_FRAME;
310         idev->rx_buff.in_frame = FALSE;
311
312         /* Test FCS and deliver frame if it’s good */
313         if (idev->rx_buff.fcs == GOOD_FCS) {
314             async_bump(idev, idev->rx_buff.data,
315                        idev->rx_buff.len);
316         } else {
317             /* Wrong CRC, discard frame!  */
318             irda_device_set_media_busy(TRUE);
319
320             DEBUG_STRING(“wrong CRC\n\r”);
321
322         }
323         break;
324     default: /* Must be the next byte of the frame */
325         if (idev->rx_buff.len < idev->rx_buff.truesize)  {
326             idev->rx_buff.data[idev->rx_buff.len++] = byte;
327             idev->rx_buff.fcs = irda_fcs(idev->rx_buff.fcs, byte);
328         } else {
329             DEBUG(1, “state_inside_frame(), Rx buffer overflow, aborting\n”);
330             idev->rx_buff.state = OUTSIDE_FRAME;
331         }
332         break;
333
334     }
335 }
```

Listing 56. Listing of wrapper.c

C.4 Debug Log Connection Establishment

This section lists the debug log for establishing an IrLAN connection between the SmartBadge and an HP Netbeam IR access point. At the end of the log the IrLAN link has been set up and is ready for data transmissions. If there are no data packets to be transmitted, the IrLAP layer constantly exchanges receiver ready messages with its peer to signal the link turnaround. If network packets have to be transmitted in either direction, they are encapsulated into info frames which can be sent until the link turnaround time has expired.

```
main: init
irlmp_init()
irlap_init()
irport_open()
irda_device_set_media_busy(FALSE)
irlap_open()
next LAP state = LAP_OFFLINE
Get saddr = 68579752
irlap_apply_default_connection_parameters()
irlap_change_speed(), setting speed to 9600
irport_change_speed SIR, 9600 baud
next LAP state = LAP_NDM
irlmp_register_link(), Registered IrLAP, saddr = 68579752
irlmp_next_lap_state(), LMP LAP = LAP_STANDBY
irport_sir_init()
iriap_init()
irlmp_register_service(), hints = 0005
irias_new_object(Device)
irias_add_string_attrib()
irias_add_string_attrib: name=DeviceName,
                         value=Badge
irias_add_attrib()
irias_insert_object()
iriap_open(), mode=IrIAS srv
irlmp_open_lsap(), slsap_sel=00
irlmp_slsap_inuse()
irlmp_slsap_inuse: is free
irlmp_next_lsap_state(), LMP LSAP = LSAP_DISCONNECTED
iriap_open(), source LSAP sel=00
iriap_next_client_state(): IAP Client=S_DISCONNECT
iriap_next_call_state(): IAP Call=S_MAKE_CALL
iriap_next_server_state(): IAP Server=R_DISCONNECT
iriap_next_r_connect_state(): R_WAITING
irttpIRLAN_IDLE
irlan_provider_open_ctrl_tsap()
irttp_open_tsap()
irlmp_open_lsap(), slsap_sel=ff
irlmp_slsap_inuse()
irlmp_slsap_inuse: is free
irlmp_find_free_slsap(), next free lsap_sel=10
irlmp_next_lsap_state(), LMP LSAP = LSAP_DISCONNECTED
irttp_open_tsap(), stsap_sel=10
irias_new_object(IrLAN)
irias_add_integer_attrib()
irias_add_attrib()
irias_insert_object()
irias_new_object(PnP)
irias_add_string_attrib()
irias_add_string_attrib: name=Name,
                         value=Badge
irias_add_attrib()
irias_add_string_attrib()
irias_add_string_attrib: name=DeviceID,
                         value=HWP19F0
irias_add_attrib()
irias_add_integer_attrib()
```

Listing 57. Debug log of connection establishment

```
irias_add_attrib()
irias_add_string_attrib()
irias_add_string_attrib: name=Comp#02,
                         value=PNP8389
irias_add_attrib()
irias_add_string_attrib()
irias_add_string_attrib: name=Manufacturer,
                         value=Badge-IrDA Project
irias_add_attrib()
irias_insert_object()
irlmp_discovery_request(), nslots=0
irlmp_discovery_request() discovery already running, so we just return the old discov-
ery log!
irlmp_discovery_confirm()
ipstack_init()
IrLMP, discovery timer expired!
irlmp_do_discovery(6)
irlmp_expire_discoveries()
irlmp_do_lap_event(), EVENT = LM_LAP_DISCOVERY_REQUEST,
                      STATE = LAP_STANDBY
irlmp_state_standby()
irlap_discovery_request(), nslots = 6
irlap_do_event(), event = DISCOVERY_REQUEST,
                  state = LAP_NDM
irlap_state_ndm()
irlap_send_discovery_xid_frame()
    s=0
    S=6
    command=1
next LAP state = LAP_QUERY
irda timer: Slot timer expired!
timer count: 1761952705
irlap_do_event(), event = SLOT_TIMER_EXPIRED,
                  state = LAP_QUERY
lap_ev slot timer exp
irlap_send_discovery_xid_frame()
    s=1
    S=6
    command=1
next LAP state = LAP_QUERY
irda timer: Slot timer expired!
timer count: 1762263679
irlap_do_event(), event = SLOT_TIMER_EXPIRED,
                  state = LAP_QUERY
lap_ev slot timer exp
irlap_send_discovery_xid_frame()
    s=2
    S=6
    command=1
next LAP state = LAP_QUERY
irda timer: Slot timer expired!
timer count: 1762574551
irlap_do_event(), event = SLOT_TIMER_EXPIRED,
                  state = LAP_QUERY
lap_ev slot timer exp
irlap_send_discovery_xid_frame()
    s=3
    S=6
    command=1
next LAP state = LAP_QUERY
irda timer: Slot timer expired!
timer count: 1762885652
irlap_do_event(), event = SLOT_TIMER_EXPIRED,
                  state = LAP_QUERY
```

Listing 57. Debug log of connection establishment

```
lap_ev slot timer exp
irlap_send_discovery_xid_frame()
    s=4
    S=6
    command=1
next LAP state = LAP_QUERY
irlap_recv_discovery_xid_rsp()
irlap_recv_discovery_xid_rsp(), daddr=000cc000,
  saddr=68579752
  info='HP NetBeamIR'.
irlap_do_event(), event = RECV_DISCOVERY_XID_RSP,
                  state = LAP_QUERY
irlap_state_query(), discovery response received, daddr=000cc000
next LAP state = LAP_QUERY
irda timer: Slot timer expired!
timer count: 1763197485
irlap_do_event(), event = SLOT_TIMER_EXPIRED,
                  state = LAP_QUERY
lap_ev slot timer exp
irlap_send_discovery_xid_frame()
    s=5
    S=6
    command=1
next LAP state = LAP_QUERY
irda timer: Slot timer expired!
timer count: 1763508766
irlap_do_event(), event = SLOT_TIMER_EXPIRED,
                  state = LAP_QUERY
lap_ev slot timer exp
lap_ev sending final slot
irlap_send_discovery_xid_frame()
    s=255
    S=6
    command=1
lap_fr queue final
next LAP state = LAP_NDM
irlap discovery confirm
irda_device_set_media_busy(FALSE)
irlmp_link_discovery_confirm()
irlmp_add_discovery_log()
irlmp_do_lap_event(), EVENT = LM_LAP_DISCOVERY_CONFIRM,
                      STATE = LAP_STANDBY
irlmp_state_standby()
irlmp_discovery_confirm()
irlmp_notify_client()
discovery->daddr = 0x000cc000
irlan_client_discovery_indication()
irlan_client_discovery_indication(), starting new instance!
irlan_open(),  register
irlan_open: switching states to IRLAN_IDLE
irlan_next_client_state(), IRLAN_IDLE
irlan_next_provider_state(), IRLAN_IDLE
irlan_register_netdev()
irlan_eth_init()
irlan_eth_init(): simulate irmanager EVENT_IRLAN_START
irmanager_notify(), event EVENT_IRLAN_START
irlan_eth_open()
irlan_client_wakeup()
irlan_client_open_ctrl_tsap()
irlan_client_open_ctrl_tsap: before irttp_open_tsap
irttp_open_tsap()
irlmp_open_lsap(), slsap_sel=ff
irlmp_slsap_inuse()
irlmp_slsap_inuse: is free
irlmp_find_free_slsap(), next free lsap_sel=11
```

Listing 57. Debug log of connection establishment

```
irlmp_next_lsap_state(), LMP LSAP = LSAP_DISCONNECTED
irttp_open_tsap(), stsap_sel=11
irlan_open_data_tsap()
irttp_open_tsap()
irlmp_open_lsap(), slsap_sel=ff
irlmp_slsap_inuse()
irlmp_slsap_inuse: is free
irlmp_find_free_slsap(), next free lsap_sel=12
irlmp_next_lsap_state(), LMP LSAP = LSAP_DISCONNECTED
irttp_open_tsap(), stsap_sel=12
irlan_client_state_idle()
iriap_getvaluebyclass_request()
iriap_open(), mode=IrIAS cli
irlmp_open_lsap(), slsap_sel=ff
irlmp_slsap_inuse()
irlmp_slsap_inuse: is free
irlmp_find_free_slsap(), next free lsap_sel=13
irlmp_next_lsap_state(), LMP LSAP = LSAP_DISCONNECTED
iriap_open(), source LSAP sel=13
iriap_next_client_state(): IAP Client=S_DISCONNECT
iriap_next_call_state(): IAP Call=S_MAKE_CALL
iriap_next_server_state(): IAP Server=R_DISCONNECT
iriap_next_r_connect_state(): R_WAITING
(iriap_event) state_s_disconnect()
iriap_next_client_state(): IAP Client=S_CONNECTING
irlmp_connect_request(), slsap_sel=13
                           dlsap_sel=00
irlmp_do_lsap_event(), EVENT = LM_CONNECT_REQUEST,
                       STATE = LSAP_DISCONNECTED
                       slsap_sel = 13
irlmp_state_disconnected()
irlmp_state_disconnected(), LM_CONNECT_REQUEST
irlmp_next_lsap_state(), LMP LSAP = LSAP_SETUP_PEND
irlmp_do_lap_event(), EVENT = LM_LAP_CONNECT_REQUEST,
                      STATE = LAP_STANDBY
irlmp_state_standby()
irlmp_state_standby() LS_CONNECT_REQUEST
irlap_connect_request(), daddr=0x000cc000
irlap_do_event(), event = CONNECT_REQUEST,
                  state = LAP_NDM
irlap_state_ndm()
irlap_send_snrm_frame()
next LAP state = LAP_SETUP
irlmp_next_lap_state(), LMP LAP = LAP_U_CONNECT
irlan_next_client_state(), IRLAN_QUERY
irlan_start_watchdog_timer()
irlan_client_start_kick_timer()
irlan_start_watchdog_timer()
UA rsp frame received!
irlap_recv_ua_frame()
irlap_do_event(), event = RECV_UA_RSP,
                  state = LAP_SETUP
irlap_state_setup()
irlap_initiate_connection_state()
irda_qos_negotiate()
Setting BAUD_RATE to 115200 bps.
Setting DATA_SIZE to 2048 bytes
Setting WINDOW_SIZE to 7
Setting XBOFS to 1
Setting MAX_TURN_TIME to 500 ms.
Setting MIN_TURN_TIME to 1000 usecs.
Setting LINK_DISC to 12 secs.
irlap_apply_connection_parameters()
irlap_change_speed(), setting speed to 115200
irport_change_speed SIR, 115200 baud
```

Listing 57. Debug log of connection establishment

```
Setting window_bytes = 5760
Setting N1 = 6
Setting N2 = 24
next LAP state = LAP_NRM_P
irlap_connect_confirm()
irlmp_link_connect_confirm()
irlmp_do_lap_event(), EVENT = LM_LAP_CONNECT_CONFIRM,
                      STATE = LAP_U_CONNECT
irlmp_state_u_connect(), event=LM_LAP_CONNECT_CONFIRM
irlmp_next_lap_state(), LMP LAP = LAP_ACTIVE
irlmp_do_lsap_event(), EVENT = LM_LAP_CONNECT_CONFIRM,
                       STATE = LSAP_SETUP_PEND
                       slsap_sel = 13
irlmp_state_setup_pend()
irlmp_send_lcf_pdu()
irlmp_next_lsap_state(), LMP LSAP = LSAP_SETUP
irda timer: Final timer expired!
irlap_do_event(), event = FINAL_TIMER_EXPIRED,
                  state = LAP_NRM_P
irlap_send_rr_frame()
irlap_state_nrm_p: FINAL_TIMER_EXPIRED: retry_count=1
irlap_recv_rr_frame()
irlap_do_event(), event = RECV_RR_RSP,
                  state = LAP_NRM_P
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
irlap_recv_rr_frame()
irlap_do_event(), event = RECV_RR_RSP,
                  state = LAP_NRM_P
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlap_do_event(), event = POLL_TIMER_EXPIRED,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=POLL_TIMER_EXPIRED
irlap_send_rr_frame()
next LAP state = LAP_NRM_P
irlap_recv_i_frame()
irlap_do_event(), event = RECV_I_RSP,
                  state = LAP_NRM_P
irlap_validate_ns_received(), as expected!
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlmp_do_lsap_event(), EVENT = LM_CONNECT_CONFIRM,
                       STATE = LSAP_SETUP
                       slsap_sel = 13
irlmp_state_setup()
irlmp_next_lsap_state(), LMP LSAP = LSAP_DATA_TRANSFER_READY
irlmp_connect_confirm()
irlmp_connect_confirm(), max_seg_size=2046
irlmp_connect_confirm(), max_header_size=4
iriap_connect_confirm()
irlmp_do_lsap_event(), EVENT = LM_DATA_REQUEST,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 13
irlmp_state_dtr()
irlap_do_event(), event = SEND_I_CMD,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
iriap_next_call_state(): IAP Call=S_OUTSTANDING
iriap_next_client_state(): IAP Client=S_CALL
```

Listing 57. Debug log of connection establishment

```
irlap_recv_rr_frame()
irlap_do_event(), event = RECV_RR_RSP,
                  state = LAP_NRM_P
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlap_do_event(), event = POLL_TIMER_EXPIRED,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=POLL_TIMER_EXPIRED
irlap_send_rr_frame()
next LAP state = LAP_NRM_P
irlap_recv_i_frame()
irlap_do_event(), event = RECV_I_RSP,
                  state = LAP_NRM_P
irlap_validate_ns_received(), as expected!
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
iriap_data_indication()
iriap_data_indication() Got ack frame!
IrLMP GetValueByClass
iriap_getvaluebyclass_confirm(), len=1
iriap_getvaluebyclass_confirm(), Value type = 1
iriap_getvaluebyclass_confirm(), lsap=1
iriap_disconnect_request()
irlmp_disconnect_request()
irlmp_do_lsap_event(), EVENT = LM_DISCONNECT_REQUEST,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 13
irlmp_state_dtr()
irlmp_send_lcf_pdu()
irlap_do_event(), event = SEND_I_CMD,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
irlmp_next_lsap_state(), LMP LSAP = LSAP_DISCONNECTED
irlmp_state_dtr(), trying to close IrLAP
irlmp_do_lap_event(), EVENT = LM_LAP_DISCONNECT_REQUEST,
                      STATE = LAP_ACTIVE
irlmp_state_active()
irlmp_state_active(), LM_LAP_DISCONNECT_REQUEST, start idle timer
irlan_client_get_value_confirm()
irlan_client_state_query()
irttp_connect_request(), max_sdu_size=1518
irlmp_connect_request(), slsap_sel=11
                         dlsap_sel=01
irlmp_do_lsap_event(), EVENT = LM_CONNECT_REQUEST,
                       STATE = LSAP_DISCONNECTED
                       slsap_sel = 11
irlmp_state_disconnected()
irlmp_state_disconnected(), LM_CONNECT_REQUEST
irlmp_next_lsap_state(), LMP LSAP = LSAP_SETUP_PEND
irlmp_do_lap_event(), EVENT = LM_LAP_CONNECT_REQUEST,
                      STATE = LAP_ACTIVE
irlmp_state_active()
irlmp_state_active(), LM_LAP_CONNECT_REQUEST
irlmp_do_lsap_event(), EVENT = LM_LAP_CONNECT_CONFIRM,
                       STATE = LSAP_SETUP_PEND
                       slsap_sel = 11
irlmp_state_setup_pend()
irlmp_send_lcf_pdu()
irlmp_next_lsap_state(), LMP LSAP = LSAP_SETUP
irlmp_do_lsap_event(), EVENT = LM_LAP_CONNECT_CONFIRM,
                       STATE = LSAP_DISCONNECTED
                       slsap_sel = 13
irlmp_state_disconnected()
```

Listing 57. Debug log of connection establishment

```
irlmp_state_disconnected(), Unknown event 13
irlmp_do_lsap_event(), EVENT = LM_LAP_CONNECT_CONFIRM,
                       STATE = LSAP_DISCONNECTED
                       slsap_sel = 12
irlmp_state_disconnected()
irlmp_state_disconnected(), Unknown event 13
irlmp_do_lsap_event(), EVENT = LM_LAP_CONNECT_CONFIRM,
                       STATE = LSAP_DISCONNECTED
                       slsap_sel = 00
irlmp_state_disconnected()
irlmp_state_disconnected(), Unknown event 13
irlmp_do_lsap_event(), EVENT = LM_LAP_CONNECT_CONFIRM,
                       STATE = LSAP_DISCONNECTED
                       slsap_sel = 10
irlmp_state_disconnected()
irlmp_state_disconnected(), Unknown event 13
irlan_next_client_state(), IRLAN_CONN
iriap_send_ack()
iriap_next_call_state(): IAP Call=S_WAIT_FOR_CALL
iriap_close()
__irlmp_close_lsap()
__iriap_close()
irlap_recv_rr_frame()
irlap_do_event(), event = RECV_RR_RSP,
                  state = LAP_NRM_P
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
irlap_recv_rr_frame()
irlap_do_event(), event = RECV_RR_RSP,
                  state = LAP_NRM_P
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlap_do_event(), event = POLL_TIMER_EXPIRED,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=POLL_TIMER_EXPIRED
irlap_send_rr_frame()
next LAP state = LAP_NRM_P
irlap_recv_i_frame()
irlap_do_event(), event = RECV_I_RSP,
                  state = LAP_NRM_P
irlap_validate_ns_received(), as expected!
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlmp_do_lsap_event(), EVENT = LM_CONNECT_CONFIRM,
                       STATE = LSAP_SETUP
                       slsap_sel = 11
irlmp_state_setup()
irlmp_next_lsap_state(), LMP LSAP = LSAP_DATA_TRANSFER_READY
irlmp_connect_confirm()
irlmp_connect_confirm(), max_seg_size=2046
irlmp_connect_confirm(), max_header_size=4
irttp_connect_confirm()
IrTTP, Negotiated BAUD_RATE: 3e
IrTTP, Negotiated BAUD_RATE: 115200 bps.
irttp_connect_confirm(), Initial send_credit=1
irttp_connect_confirm(), RxMaxSduSize=1518
irlan_client_ctrl_connect_confirm()
irlan_client_state_conn()
irlan_get_provider_info()
irlan_ctrl_data_request()
irlan_ctrl_data_request() running irlan_run_ctrl_tx_queue
irlan_run_ctrl_tx_queue()
```

Listing 57. Debug log of connection establishment

```
irttp_data_request()
irlmp_do_lsap_event(), EVENT = LM_DATA_REQUEST,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 11
irlmp_state_dtr()
irlap_do_event(), event = SEND_I_CMD,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
irlan_next_client_state(), IRLAN_INFO
irlap_recv_rr_frame()
irlap_do_event(), event = RECV_RR_RSP,
                  state = LAP_NRM_P
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlap_do_event(), event = POLL_TIMER_EXPIRED,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=POLL_TIMER_EXPIRED
irlap_send_rr_frame()
next LAP state = LAP_NRM_P
irlap_recv_i_frame()
irlap_do_event(), event = RECV_I_RSP,
                  state = LAP_NRM_P
irlap_validate_ns_received(), as expected!
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlan_client_ctrl_data_indication()
irlan_client_state_info()
irlan_client_parse_response() skb->len=29
irlan_client_parse_response(), got 2 parameters
irlan_extract_param()
Parameter: MEDIA
Value: 802.3
irlan_check_response_param(), parm=MEDIA
irlan_extract_param()
Parameter: IRLAN_VER
Value:
irlan_check_response_param(), parm=IRLAN_VER
IrLAN version 1
              .1
irlan_next_client_state(), IRLAN_MEDIA
irlan_get_media_char()
irlan_ctrl_data_request()
irlan_ctrl_data_request() running irlan_run_ctrl_tx_queue
irlan_run_ctrl_tx_queue()
Trying to lock already locked variable!
  irlan_ctrl_data_request: error value=-16
irlan_client_ctrl_data_indication running irlan_run_ctrl_tx_queue
irlan_run_ctrl_tx_queue()
irttp_data_request()
irlmp_do_lsap_event(), EVENT = LM_DATA_REQUEST,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 11
irlmp_state_dtr()
irlap_do_event(), event = SEND_I_CMD,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
IrLMP, discovery timer expired!
irlmp_do_discovery(6)
irlap_recv_rr_frame()
irlap_do_event(), event = RECV_RR_RSP,
                  state = LAP_NRM_P
```

Listing 57. Debug log of connection establishment

```
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlap_do_event(), event = POLL_TIMER_EXPIRED,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=POLL_TIMER_EXPIRED
irlap_send_rr_frame()
next LAP state = LAP_NRM_P
irlap_recv_i_frame()
irlap_do_event(), event = RECV_I_RSP,
                  state = LAP_NRM_P
irlap_validate_ns_received(), as expected!
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlan_client_ctrl_data_indication()
irlan_client_state_media()
irlan_client_parse_response() skb->len=104
irlan_client_parse_response(), got 5 parameters
irlan_extract_param()
Parameter: FILTER_TYPE
Value: DIRECTED
irlan_check_response_param(), parm=FILTER_TYPE
irlan_extract_param()
Parameter: FILTER_TYPE
Value: BROADCAST
irlan_check_response_param(), parm=FILTER_TYPE
irlan_extract_param()
Parameter: FILTER_TYPE
Value: MULTICAST
irlan_check_response_param(), parm=FILTER_TYPE
irlan_extract_param()
Parameter: ACCESS_TYPE
Value: DIRECT
irlan_check_response_param(), parm=ACCESS_TYPE
irlan_extract_param()
Parameter: MAX_FRAME
Value: n
irlan_check_response_param(), parm=MAX_FRAME
irlan_check_response_param(), max frame=1518
irlan_open_data_channel()
irlan_ctrl_data_request()
irlan_ctrl_data_request() running irlan_run_ctrl_tx_queue
irlan_run_ctrl_tx_queue()
Trying to lock already locked variable!
  irlan_ctrl_data_request: error value=-16
irlan_next_client_state(), IRLAN_OPEN
irlan_client_ctrl_data_indication running irlan_run_ctrl_tx_queue
irlan_run_ctrl_tx_queue()
irttp_data_request()
irlmp_do_lsap_event(), EVENT = LM_DATA_REQUEST,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 11
irlmp_state_dtr()
irlap_do_event(), event = SEND_I_CMD,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
irlap_recv_rr_frame()
irlap_do_event(), event = RECV_RR_RSP,
                  state = LAP_NRM_P
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlmp_do_lap_event(), EVENT = LM_LAP_IDLE_TIMEOUT,
                      STATE = LAP_ACTIVE
irlmp_state_active()
```

Listing 57. Debug log of connection establishment

```
irlmp_state_active(), IDLE_TIMEOUT, lsaps!=0
irlap_do_event(), event = POLL_TIMER_EXPIRED,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=POLL_TIMER_EXPIRED
irlap_send_rr_frame()
next LAP state = LAP_NRM_P
irlap_recv_i_frame()
irlap_do_event(), event = RECV_I_RSP,
                  state = LAP_NRM_P
irlap_validate_ns_received(), as expected!
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlan_client_ctrl_data_indication()
irlan_client_state_open()
irlan_client_parse_response() skb->len=39
irlan_client_parse_response(), got 2 parameters
irlan_extract_param()
Parameter: DATA_CHAN
Value:
irlan_check_response_param(), parm=DATA_CHAN
Data TSAP = 02
irlan_extract_param()
Parameter: RECONNECT_KEY
Value:
irlan_check_response_param(), parm=RECONNECT_KEY
Got reconnect key:
irttp_connect_request(), max_sdu_size=1518
irlmp_connect_request(), slsap_sel=12
                           dlsap_sel=02
irlmp_do_lsap_event(), EVENT = LM_CONNECT_REQUEST,
                       STATE = LSAP_DISCONNECTED
                       slsap_sel = 12
irlmp_state_disconnected()
irlmp_state_disconnected(), LM_CONNECT_REQUEST
irlmp_next_lsap_state(), LMP LSAP = LSAP_SETUP_PEND
irlmp_do_lap_event(), EVENT = LM_LAP_CONNECT_REQUEST,
                      STATE = LAP_ACTIVE
irlmp_state_active()
irlmp_state_active(), LM_LAP_CONNECT_REQUEST
irlmp_do_lsap_event(), EVENT = LM_LAP_CONNECT_CONFIRM,
                       STATE = LSAP_SETUP_PEND
                       slsap_sel = 12
irlmp_state_setup_pend()
irlmp_send_lcf_pdu()
irlap_do_event(), event = SEND_I_CMD,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
irlmp_next_lsap_state(), LMP LSAP = LSAP_SETUP
irlmp_do_lsap_event(), EVENT = LM_LAP_CONNECT_CONFIRM,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 11
irlmp_state_dtr()
irlmp_state_dtr(), Unknown event 13
irlmp_do_lsap_event(), EVENT = LM_LAP_CONNECT_CONFIRM,
                       STATE = LSAP_DISCONNECTED
                       slsap_sel = 00
irlmp_state_disconnected()
irlmp_state_disconnected(), Unknown event 13
irlmp_do_lsap_event(), EVENT = LM_LAP_CONNECT_CONFIRM,
                       STATE = LSAP_DISCONNECTED
                       slsap_sel = 10
irlmp_state_disconnected()
irlmp_state_disconnected(), Unknown event 13
```

Listing 57. Debug log of connection establishment

```
irlan_next_client_state(), IRLAN_DATA
irlan_client_ctrl_data_indication running irlan_run_ctrl_tx_queue
irlan_run_ctrl_tx_queue()
irttp_todo_expired()
   irttp queues run
irlap_recv_rr_frame()
irlap_do_event(), event = RECV_RR_RSP,
                  state = LAP_NRM_P
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlap_do_event(), event = POLL_TIMER_EXPIRED,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=POLL_TIMER_EXPIRED
irlap_send_rr_frame()
next LAP state = LAP_NRM_P
irlap_recv_i_frame()
irlap_do_event(), event = RECV_I_RSP,
                  state = LAP_NRM_P
irlap_validate_ns_received(), as expected!
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlmp_do_lsap_event(), EVENT = LM_CONNECT_CONFIRM,
                       STATE = LSAP_SETUP
                       slsap_sel = 12
irlmp_state_setup()
irlmp_next_lsap_state(), LMP LSAP = LSAP_DATA_TRANSFER_READY
irlmp_connect_confirm()
irlmp_connect_confirm(), max_seg_size=2046
irlmp_connect_confirm(), max_header_size=4
irttp_connect_confirm()
IrTTP, Negotiated BAUD_RATE: 3e
IrTTP, Negotiated BAUD_RATE: 115200 bps.
irttp_connect_confirm(), Initial send_credit=12
irttp_connect_confirm(), RxMaxSduSize=1518
irlan_connect_confirm()
IrLAN, We are now connected!
irlan_get_unicast_addr()
irlan_ctrl_data_request()
irlan_ctrl_data_request() running irlan_run_ctrl_tx_queue
irlan_run_ctrl_tx_queue()
irttp_data_request()
irlmp_do_lsap_event(), EVENT = LM_DATA_REQUEST,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 11
irlmp_state_dtr()
irlap_do_event(), event = SEND_I_CMD,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
irlap_recv_rr_frame()
irlap_do_event(), event = RECV_RR_RSP,
                  state = LAP_NRM_P
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlap_do_event(), event = POLL_TIMER_EXPIRED,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=POLL_TIMER_EXPIRED
irlap_send_rr_frame()
next LAP state = LAP_NRM_P
irlap_recv_i_frame()
irlap_do_event(), event = RECV_I_RSP,
                  state = LAP_NRM_P
irlap_validate_ns_received(), as expected!
irlap_validate_nr_received(), as expected!
```

Listing 57. Debug log of connection establishment

```
next LAP state = LAP_XMIT_P
irlan_client_ctrl_data_indication()
irlan_client_state_data()
irlan_client_parse_response() skb->len=55
irlan_client_parse_response(), got 3 parameters
irlan_extract_param()
Parameter: FILTER_MODE
Value: NONE
irlan_check_response_param(), parm=FILTER_MODE
irlan_extract_param()
Parameter: MAX_ENTRY
Value:
irlan_check_response_param(), parm=MAX_ENTRY
irlan_extract_param()
Parameter: FILTER_ENTRY
Value:
irlan_check_response_param(), parm=FILTER_ENTRY
Got ethernet address (FILTER_ENTRY).

irlan_client_ctrl_data_indication running irlan_run_ctrl_tx_queue
irlan_run_ctrl_tx_queue()
irttp_todo_expired()
   irttp queues run
Angel_NetstartMain() executed, IP/UDP stack initialized, now setting filters
irlan_open_unicast_addr()
irlan_ctrl_data_request()
irlan_ctrl_data_request() running irlan_run_ctrl_tx_queue
irlan_run_ctrl_tx_queue()
irttp_data_request()
irlmp_do_lsap_event(), EVENT = LM_DATA_REQUEST,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 11
irlmp_state_dtr()
irlap_do_event(), event = SEND_I_CMD,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
irlan_s~et_b>roadcast@#$_filter(), status=TRUE
irlan_ctrl_data_request()
irlan_ctrl_data_request() running irlan_run_ctrl_tx_queue
irlan_run_ctrl_tx_queue()
Trying to lock already locked variable!
  irlan_ctrl_data_request: error value=-16
irlan_set_multicast_filter(), status=FALSE
irlan_ctrl_data_request()
irlan_ctrl_data_request() running irlan_run_ctrl_tx_queue
irlan_run_ctrl_tx_queue()
Trying to lock already locked variable!
  irlan_ctrl_data_request: error value=-16
unicast and broadcast enabled, multicast disabled, ready to go
irla~>n_eth_xmit() called, size=342
irttp_data_request()
irlmp_do_lsap_event(), EVENT = LM_DATA_REQUEST,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 12
irlmp_state_dtr()
   irlan_eth_xmit() packet sent.
irlap_recv_rr_frame()
irlap_do_event(), event = RECV_RR_RSP,
                  state = LAP_NRM_P
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
```

Listing 57. Debug log of connection establishment

```
next LAP state = LAP_NRM_P
irlap_recv_i_frame()
irlap_do_event(), event = RECV_I_RSP,
                  state = LAP_NRM_P
irlap_validate_ns_received(), as expected!
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlan_client_ctrl_data_indication()
irlan_client_state_data()
irlan_client_parse_response() skb->len=2
irlan_client_parse_response(), got 0 parameters
irlan_client_ctrl_data_indication running irlan_run_ctrl_tx_queue
irlan_run_ctrl_tx_queue()
irttp_data_request()
irlmp_do_lsap_event(), EVENT = LM_DATA_REQUEST,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 11
irlmp_state_dtr()
irlap_do_event(), event = SEND_I_CMD,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
irlap_recv_i_frame()
irlap_do_event(), event = RECV_I_RSP,
                  state = LAP_NRM_P
irlap_validate_ns_received(), as expected!
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlan_eth_receive()
Angel_AngelConfigureIP() executed, IP address is 130.237.15.251
irlan_eth_xmit() called, size=42
irttp_data_request()
irlmp_do_lsap_event(), EVENT = LM_DATA_REQUEST,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 12
irlmp_state_dtr()
irlap_do_event(), event = SEND_I_CMD,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
   irlan_eth_xmit() packet sent.
irlap_recv_i_frame()
irlap_do_event(), event = RECV_I_RSP,
                  state = LAP_NRM_P
irlap_validate_ns_received(), as expected!
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlan_client_ctrl_data_indication()
irlan_client_state_data()
irlan_client_parse_response() skb->len=2
irlan_client_parse_response(), got 0 parameters
irlan_client_ctrl_data_indication running irlan_run_ctrl_tx_queue
irlan_run_ctrl_tx_queue()
irttp_data_request()
irlmp_do_lsap_event(), EVENT = LM_DATA_REQUEST,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 11
irlmp_state_dtr()
irlap_do_event(), event = SEND_I_CMD,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
```

Listing 57. Debug log of connection establishment

```
irlap_recv_i_frame()
irlap_do_event(), event = RECV_I_RSP,
                  state = LAP_NRM_P
irlap_validate_ns_received(), as expected!
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlan_eth_receive()
irlan_eth_xmit() called, size=71
irttp_data_request()
irlmp_do_lsap_event(), EVENT = LM_DATA_REQUEST,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 12
irlmp_state_dtr()
irlap_do_event(), event = SEND_I_CMD,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
   irlan_eth_xmit() packet sent.
irlap_recv_i_frame()
irlap_do_event(), event = RECV_I_RSP,
                  state = LAP_NRM_P
irlap_validate_ns_received(), as expected!
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlan_client_ctrl_data_indication()
irlan_client_state_data()
irlan_client_parse_response() skb->len=2
irlan_client_parse_response(), got 0 parameters
irlan_client_ctrl_data_indication running irlan_run_ctrl_tx_queue
irlan_run_ctrl_tx_queue()
irttp_todo_expired()
   irttp queues run
irlan_eth_xmit() called, size=71
irttp_data_request()
irlmp_do_lsap_event(), EVENT = LM_DATA_REQUEST,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 12
irlmp_state_dtr()
irlap_do_event(), event = SEND_I_CMD,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
   irlan_eth_xmit() packet sent.
irlan_eth_xmit() called, size=71
irttp_data_request()
irlmp_do_lsap_event(), EVENT = LM_DATA_REQUEST,
                       STATE = LSAP_DATA_TRANSFER_READY
                       slsap_sel = 12
irlmp_state_dtr()
   irlan_eth_xmit() packet sent.
irlap_recv_rr_frame()
irlap_do_event(), event = RECV_RR_RSP,
                  state = LAP_NRM_P
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlap_state_xmit_p(), event=SEND_I_CMD
irlap_send_i_frame()
next LAP state = LAP_NRM_P
irlap_recv_i_frame()
irlap_do_event(), event = RECV_I_RSP,
                  state = LAP_NRM_P
irlap_validate_ns_received(), as expected!
irlap_validate_nr_received(), as expected!
```

Listing 57. Debug log of connection establishment

```
next LAP state = LAP_XMIT_P
IrLMP, discovery timer expired!
irlmp_do_discovery(6)
irlap_do_event(), event = POLL_TIMER_EXPIRED,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=POLL_TIMER_EXPIRED
irlap_send_rr_frame()
next LAP state = LAP_NRM_P
irlap_recv_rr_frame()
irlap_do_event(), event = RECV_RR_RSP,
                  state = LAP_NRM_P
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlap_do_event(), event = POLL_TIMER_EXPIRED,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=POLL_TIMER_EXPIRED
irlap_send_rr_frame()
next LAP state = LAP_NRM_P
irlap_recv_rr_frame()
irlap_do_event(), event = RECV_RR_RSP,
                  state = LAP_NRM_P
irlap_validate_nr_received(), as expected!
next LAP state = LAP_XMIT_P
irlap_do_event(), event = POLL_TIMER_EXPIRED,
                  state = LAP_XMIT_P
irlap_state_xmit_p(), event=POLL_TIMER_EXPIRED
irlap_send_rr_frame()
next LAP state = LAP_NRM_P
```

Listing 57. Debug log of connection establishment