Implementation and integration of the IrDA Data Protocols into the Telelogic Tau TTCN Suite

Mats Andersson

Civilingenjörsprogrammet Datateknik
Institutionen för Systemteknik
Avdelningen för Datorkommunikation
Implementation and integration of the IrDA Data Protocols into the Telelogic Tau TTCN Suite

Mats Andersson

Ericsson Wireless Research and
LuleåUniversity of Technology
Department of Computer Science
Abstract

The use of efficient testing tools can shorten the actual testing time in projects. Ericsson Wireless Research in Dallas is evaluating the use for such tools. One of the areas where it is of interest is AXE service testing. The idea is to test AXE services via cell phones, connected and controlled from a workstation running the test tools. The cell phone in question is the Ericsson CF-888, which has a built-in modem which can be controlled via a serial port using simple AT commands.

Because of a flaw in the current pre-release of the CF-888 (as of when this is written) full functionality can not be obtained when the phone is controlled with AT-commands directly over a serial link. The solution was to first process the AT-commands with the IrDA protocol stack, and send the IrDA frames as pure data over the serial link. This works because the phone will pass any incoming unknown data to be processed by its internal IrDA protocol stack.

The main goal of this thesis was to implement the IrDA protocol stack on a Sun Solaris Workstation to be able to communicate with the CF-888 IrDA protocol stack, and by that control the phone. In this way, full functionality can be obtained, even though the phone is controlled over a serial link.

A library of abstract tests for the CF-888 was also created. The purpose of these test steps was to build a basic foundation of test building blocks that future tests can be established from. The idea is simple enough: reuse of code.
Preface

This thesis was made at Ericsson Wireless Research in Dallas from August to December 1999.

The author would like to thank the following people:

- Peter Hjelm for support during the time at Ericsson
- Thomas Nilsson for support during the time at Ericsson
- Jenny Choudhury for all guidance in the TTCN world
- Björn Nordgren, Supervisor at Luleå University of Technology, for being around
- Anneli Johansson for patience in Dallas
### Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI</td>
<td>American National Standard Institute</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AT</td>
<td>ATtention</td>
</tr>
<tr>
<td>ATS</td>
<td>Abstract Test Suite</td>
</tr>
<tr>
<td>BOF</td>
<td>Beginning Of Frame</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>DECT</td>
<td>Digital European Cordless Telephony</td>
</tr>
<tr>
<td>EOF</td>
<td>End Of Frame</td>
</tr>
<tr>
<td>ETS</td>
<td>Executable Test Suite</td>
</tr>
<tr>
<td>ESSI</td>
<td>Ericsson Strategic Software Initiative</td>
</tr>
<tr>
<td>ETSI</td>
<td>European Telecommunications Standards Institute</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>GSM</td>
<td>Global System for Mobile Communications</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IOG</td>
<td>Input/Output Group</td>
</tr>
<tr>
<td>IrCOMM</td>
<td>IrDA Serial and Parallel Port emulation</td>
</tr>
<tr>
<td>IrDA</td>
<td>Infrared Data Association</td>
</tr>
<tr>
<td>IrIAS</td>
<td>IrDA Information Access Service</td>
</tr>
<tr>
<td>IrLAP</td>
<td>IrDA Link Access Protocol</td>
</tr>
<tr>
<td>IrLMP</td>
<td>IrDA Link Management Protocol</td>
</tr>
<tr>
<td>IrTTP</td>
<td>IrDA Tiny Transport Protocol</td>
</tr>
<tr>
<td>ISDN</td>
<td>Integrated Services Digital Network</td>
</tr>
<tr>
<td>ITEX</td>
<td>Interactive TTCN Editor and eXecutor</td>
</tr>
<tr>
<td>IUT</td>
<td>Implementation Under Test</td>
</tr>
<tr>
<td>LM-MUX</td>
<td>Link Management Multiplier</td>
</tr>
<tr>
<td>MCS</td>
<td>Man machine Communication Subsystem interface</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnection</td>
</tr>
<tr>
<td>SDL</td>
<td>Specification Description Language</td>
</tr>
<tr>
<td>SMS</td>
<td>Short Message Service</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TTCN</td>
<td>Tree and Tabular Combined language</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>
CONTENTS

Preface ................................ ................................ ................................ ...................... 3

1 Introduction ................................ ................................ ................................ ....... 7

2 Introduction to the IrDA Data Protocols ................................ ......................... 12

2.1 Mandatory IrDA Data Protocols ................................................................. 12

2.1.1 Physical Layer ........................................................................................... 12

2.1.2 IrLAP – IrDA Link Access Protocol ......................................................... 12

2.1.2.1 IrLAP Modes .......................................................................................... 13

2.1.2.2 IrLAP Services ....................................................................................... 13

2.1.2.3 IrLAP Frame Format ............................................................................. 14

2.1.3 IrLMP – IrDA Link Management Protocol........................................... 14

2.1.3.1 Link Management Components ......................................................... 15

2.1.3.2 IAS – Information Access Service ................................................... 15

2.1.3.3 LM-MUX .............................................................................................. 15

2.1.3.4 IrLMP Services .................................................................................... 16

2.1.3.5 IrLMP Frame Format ........................................................................ 16

2.2 Optional IrDA Data Protocols ..................................................................... 17

2.2.1 IrTTP – IrDA Tiny Transport Protocol .................................................. 17

2.2.1.1 IrTTP Flow Control ............................................................................ 18

2.2.1.2 IrTTP Frame Format ........................................................................ 18

2.2.2 IrCOMM – IrDA Serial and Parallel Port emulation ......................... 18

2.2.2.1 3-Wire Raw .......................................................................................... 19

2.2.2.2 3-Wire ................................................................................................. 19

2.2.2.3 9-Wire ................................................................................................. 19

2.2.2.4 IrCOMM Service Interface ................................................................. 19

2.2.2.5 IrCOMM Frame Formats ................................................................ 20

3 Implementation of the IrDA Data Protocols .............................................. 22

3.1 General .......................................................................................................... 22

3.2 System design idea ......................................................................................... 22

3.2.1 Integration Idea ......................................................................................... 23

3.3 The IrDA Data Protocols ............................................................................. 25

3.3.1 Overall Protocol Stack Module Design ................................................. 25

3.3.2 Physical Layer Implementation ............................................................... 27

3.3.3 IrLAP Implementation .............................................................................. 28

3.3.4 IrLMP Implementation .......................................................................... 29

3.3.5 IAS Implementation ............................................................................... 29

3.3.6 IrTTP Implementation ........................................................................... 30

3.3.7 IrCOMM Implementation ....................................................................... 31

3.3.8 Port Emulation Implementation .............................................................. 33

3.3.9 Control Module Implementation ........................................................... 34

3.3.10 Integration .............................................................................................. 35

3.3.11 Special Porting Difficulties .................................................................... 35

3.3.11.1 Memory Handling ............................................................................ 35
3.4 Evaluation of the IrDA Implementation ................................................................. 36

4 TTCN Test Case Implementation ............................................................... 38

4.1 Test Step Library .......................................................................................... 38

4.1.1 INIT Group .................................................................................................. 38

4.1.1.1 Remove Pending Input .............................................................................. 38

4.1.1.2 Init CF888 ................................................................................................. 38

4.1.2 CONTROL Group ........................................................................................ 38

4.1.2.1 CMD Test Step .......................................................................................... 39

4.1.3 VOICE CALL Group ......................................................................................... 39

4.1.3.1 Voice Call Up ............................................................................................... 39

4.1.3.2 Voice Call With Time .................................................................................. 39

4.1.3.3 Voice Hang Up .............................................................................................. 39

4.1.4 DATA CALL Group ........................................................................................ 39

4.1.4.1 Data Call Up ............................................................................................... 39

4.1.4.2 Data Hang Up .............................................................................................. 39

4.1.4.3 Send Data ................................................................................................. 39

4.1.5 SMS Group .................................................................................................. 39

4.1.5.1 Set SMSC Address .................................................................................... 39

4.1.5.2 SMS Store .................................................................................................. 40

4.1.5.3 SMS Send ................................................................................................... 40

4.2 Evaluation of the TTCN Test Step Library .................................................. 40

5 Result ............................................................................................................... 41

5.1 Further Work .................................................................................................. 41

References ............................................................................................................ 43
1 Introduction

1.1 Background

One part of the Ericsson Strategic Software Initiative (ESSI) project at Ericsson Wireless Research has the purpose to evaluate efficient testing tools in order to shorten the actual testing time in projects. The criterion is that these testing tools are to be using formal methods.

A commonly used definition of formal methods is as follows:

A formal method in software development is a method that provides a formal language for describing a software artifact (e.g. specifications, design, source-code) such that formal proofs are possible, in principle, about properties of the artifact so expressed.

The internal test and verification department at Ericsson Wireless Research is interested in finding such formal test tools to accomplish automated, more effective testing. One area where such automated testing would be of interest is AXE service testing. A pre-evaluation of the Telelogic Tau TTCN suite, indicated that the set was suitable for such testing, and that a license should be purchased for the purpose of a further evaluation of the tool set.

1.1.1 Description of the Telelogic Tau TTCN Suite

The Telelogic Tau TTCN Suite is part of the Telelogic Tau environment, which is an environment for visualizing, developing, implementing and testing reliable real-time systems software. The environment offers specialized tool sets for every phase of a project: Telelogic Tau UML Suite for requirement capture and analysis, Telelogic Tau SDL Suite for design and implementation, and Telelogic Tau TTCN Suite for comprehensive testing [9].

The full power of the environment comes when the different tool sets are used together, interactively between the different project phases. An example of this is the SDL to TTCN Link, which allows automatic test case generation from SDL specifications. This enables the developer to consistence check his final application with the original specific actions.

At the time being there was no practical way to evaluate the tool suite as a whole. The different tool sets was to be evaluated individually by the corresponding departments, which is why the Telelogic Tau TTCN Suite was to be evaluated by the test and verification department as part of the ESSI project.

1.1.1.1 TTCN

TTCN (Tree and Tabular Combined Language) is a formal language for the specification of test cases. It was originally determined for the definition of test suites for the OSI protocol stack. TTCN is defined in part 3 of the ISO standard 9646.

TTCN has essentially been used to define standards. It is for example often used by ETSI (European Telecommunications Standards Institute) for the definition of conformance test suites for the European telecom standards, e.g. GSM, DECT and ISDN. It is however suitable for all kinds of functional testing for real-time and communication systems. This has led to a wide usage throughout the telecommunications industry.
TTCN is a system independent language. This means that a test suite in TTCN for one application (e.g., protocol, system) can be used in any test environment for that application.

Figure 1 in the next section shows a test written in TTCN with the tool ITEX. ITEX is an abbreviation for Interactive TTCN Editor and eXecutor. It is a family of tools for the development, specification, and compilation of executable tests for the TTCN language [9]. With ITEX graphical environment, TTCN test scripts are created with the TTCN editor, and checked for syntactic and semantic errors by means of a TTCN analyzer.

The tests are organized in a hierarchy consisting of three levels. At the lowest level of abstraction are the test steps. Next are the test cases, which can call the existing test steps, and last, test cases are organized in abstract test suites. Figure 1 shows a typical test case, written in the TTCN editor. Notice how the Test step REMOVE_PENDING is called in the first two lines in the Test case VOICE_CALL_UP.

By means of ITEX' TTCN compiler, automatic translation from TTCN to ANSI-C is feasible. In this way an ETS (Executable Test Suite) can be generated from anyATS (Abstract Test Suite). In order to be executable, the compiled code must be linked to a library containing the necessary operating system functions.

### 1.1.1.2 Naute Test System

The Naute Test System is a distributed test environment intended for AXE service testing. It is supposed to work as a stand-alone platform, for testing, or as a complement to the ITEX TTCN Editor and the ITEX C Code Generator for test development. The test platform basically consists of a number of AXE stations on which workstations are connected both by means of telephones (modem etc) as well as by serial cables to MCS (Man machine Communication Subsystem interface). MCS is a command interface to AXE switches.

<table>
<thead>
<tr>
<th>Step</th>
<th>Event</th>
<th>Description</th>
<th>Constraints Set</th>
<th>E值</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>REMOVE_PENDING(200s),</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>REMOVE_PENDING(20s),</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>COMMAND(861) T/D(20s),</td>
<td>VOICE DIAL(50s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>RES_ID(15s)</td>
<td>RING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>COMMAND(862) T/D(5s)</td>
<td>COMMAND(862) T/D(5s)</td>
<td>RING</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>to 1 RESPONSE</td>
<td>RING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>COMMAND(863) T/D(5s)</td>
<td>COMMAND(863) T/D(5s)</td>
<td>RING</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>to 1 RESPONSE CANCEL T/D (OK</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Naute GUI:
The Naute GUI is used to execute the ETS generated by ITEX. On running the Naute GUI, the Naute Log and the Test Selector windows pops up on the screen. Test cases can be selected and executed from the test Test Selector, which also gives us the result: one of the three possible verdicts - PASS, FAIL and INCONCLUSIVE. A complete step by step description of the execution of the test cases will be written in Naute Log.

Naute Test System Overview:
The Naute Test System provides the distributed test environment shown in figure 2.

Workstation environment

Test Operator Workstation
ITEX (Editing) Naute GUI (Execution)

Test PCO Server
Workstation
mkserver (tcp/ip)
modem (AT modem com.)
iog-11 (IOG-11 com.)

TCP/IP

IUT
(Implementation Under Test)

By means of a TCP/IP network (Ethernet etc) the workstation environment can be split over several connected workstations.

The server software (mkserver) is a simple standard input and output redirection program. It creates a TCP/IP socket and listens for incoming connections from the Naute GUI. When a connection is made it starts a communication program as a subprocess. The subprocess standard input and output is transferred via the TCP/IP socket. Each server handles one connection to a specific entity. If the executed test case involves three phones, three servers must be started to handle the communications. To be able to communicate with different kind of entities, the subprocess must start different kinds of communications program. The Naute platform (as of when this is written) provides two types of communications programs – “modem” which connects to an AT compatible modem, and iog-11 which communicates with an iog-7 or iog-11 class MCS type communications port of an AXE switch. The AT command set is the universal standard for sending commands to modems.

The ability to communicate directly with the AXE via the MCS interface was according to [10] an important factor for choosing the tool suite. By means of the MSC interface the AXE switch can be monitored and controlled directly. This makes it possible to check the AXE status as the phones is using its services.

The IUT (Implementation under test) in figure 2 is the tested implementation. Here this is the AXE services. The IUT can be accessed directly with the MSC interface connected via a serial RS-232 link, or indirectly via the mobile phones connected via a serial RS-232 link.
1.1.2 The Ericsson CF-888 Cell Phone

As mentioned, the Naute test suite can only communicate in two ways, of which one is with standard AT-compatible modems. This is a restraint on the testing possibilities. The dependency on AT-compatible modems brings, for example, the issue that speech testing is hard, if not impossible to achieve. In [10], Peter Hjelm concludes that since the test and verification department is interested in finding out how to do effective automated testcases where speech is not the major factor, the Telelogic test suite is sufficient for most cases. However, to access the AXE services through AT-compatible modems, the only option was to use mobile phones with built in modems.

The phone used is the Ericsson CF-888 that has a built in modem. The CF-888 has got two interfaces, an infrared eye interface and a serial port interface. Both can be used to control the phone with AT-commands. However, during testing a serial link is to prefer over an infrared link due to its more stable nature. Further, by using a serial port multiplier, multiple phones can be connected and controlled from one single workstation.

1.1.2.1 The CF-888 flaw

In the pre-evaluation of the Telelogic Tau TTCN Suite, it was discovered that when the CF-888 was controlled directly over a serial link, some AT-commands did not work properly. In fact, the phone rejected them as erroneous commands. By investigating the problem it was discovered that it was due to a flaw in the phone processor. The Ericsson Mobile department in Lund indicated that this problem could be avoided by sending the commands as IrDA Data frames over the serial link. This would work because of the phone’s way to handle incoming data on the serial link.

The phone has actually got two processors – the main telephone processor and the modem processor. The infrared communication through the phone’s infrared eye interface is handled by the modem processor, which has got support for the IrDA protocol stack. Now, all incoming data on the serial link is checked for its “type”. So, if incoming data on the serial link is discovered to be IrDA data, it is passed to the modem processor to be processed by the IrDA stack. In this way it would be possible to obtain full control over the phone even though it is controlled over a serial link.

1.2 Goals

The main goal of this thesis was to implement the IrDA Data Protocols on a Sun Solaris Unix Workstation, and further integrate the protocol stack in the existing test platform Naute. The purpose was to obtain full control over the Ericsson CF-888 mobile phone, by means of AT-commands. This would keep the originally intended system setup with multiple phones controlled from a single workstation.

Figure 3 shows a possible physical setup of the intended test system. Notice that the phones are connected via physical cables to the serial ports. The serial multiplier used has 16 interfaces, which enables a maximum of 15 mobile phones to be connected as one must be reserved for communicating directly with the switch through the MSC interface.
The secondary goal of this thesis was to write a library of TTCN Test Cases in ITEX. The purpose was to form a foundation for future Test Case writing. The Test Cases is written in the scope of the CF-888 phone as the controlled entities.

**Physical Setup**

*Figure 3. Example of a physical setup of the Naute Test System.*
2 Introduction to the IrDA Data Protocols

The Infrared Data Association (IrDA) is an international association of companies focused on providing infrared (IR) standards to ensure the quality and interoperability of the Infrared Technology [5].

“IrDA Data” defines a standard for cordless infrared light data transfer. It is recommended for high-speed short range, line of sight, point-to-point communication—suitable for laptops, digital cameras, handheld data collection devices, etc.

Figure 4 shows the IrDA Data Protocols arranged as a stack. The layers within this stack can be divided into two groups – mandatory (shaded) and optional protocols.

2.1 Mandatory IrDA Data Protocols

The mandatory layers of the IrDA protocol stack include the following:

- Physical layer
- IrLAP: Link Access Protocol
- IrLMP: Link Management Protocol

2.1.1 Physical Layer

The physical layer handles the encoding and signaling of data bits. It includes (normally) the optical transceiver, and deals with shaping and other characteristics of infrared signals, framing of data such as end of frame flags (BOFs and EOFs) and cyclic redundancy checks (CRCs). It is partially or entirely handled by hardware.

2.1.2 IrLAP – IrDA Link Access Protocol

IrLAP corresponds to OSI layer 2 (data link protocol). It provides point-to-point connections (one-to-one).

IrLAP provides reliable data transfer using the following mechanisms:

- Retransmission.
• Low-level flow control.
• Error detection.

The flow control uses mechanism similar to the sliding window mechanism used in TCP. The IrDA mechanism is however not as flexible (only a maximum of seven unacknowledged frames can be sent) and comprehensive as the TCP mechanism.

The two parties within an IrLAP connection have a master-slave relationship with different responsibilities. The IrDA terms for this is Primary (master) and Secondary (slave).

**Primary Station:**
• Sends Command frames – initiates connections and transfers.
• Responsible for organization and control of data flow.
• Deals with unrecoverable data link errors.

**Secondary Station:**
• Sends response frames – only speaks when spoken to.

### 2.1.2.1 IrLAP Modes
IrLAP is built around two modes, corresponding to whether or not a connection exists – Normal Disconnect Mode (NDM) and Normal Response Mode (NRM). In NDM no connection has been established. NRM is the mode of operation for connected devices. In NDM all devices have the same default link settings. This is because the devices must be able to find each other first, before a connection can be established. During the connection process, the two sides exchange capability information in order to create the best connection possible that is supported by both sides. The IrLAP operating procedures for the two modes are specified in [1].

### 2.1.2.2 IrLAP Services
In [1] the services provided by the IrLAP layer to the upper layers are described. The services are specified as service primitives - an abstraction that specify only the service provided rather than the means by which the service is provided. Not all services are necessary for all devices. The most important services include the following:

• Device Discovery: Explores the nearby IR-space to see who is present and get some hint what capabilities they have.
• Connect: Chooses a specific partner, negotiates the best possible communications parameters supported by both sides, and connects.
• Send Data: Data can either be sent as reliable sequenced data or unreliable unsequenced data. Used by higher layer protocols for almost all of their work.
• Disconnect: Closes down and returns to the NDM state ready for a new connection.
2.1.2.3 IrLAP Frame Format

The basic IrLAP frame is as follows:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Control</td>
<td>Information</td>
</tr>
<tr>
<td>8 bits</td>
<td>8 bits</td>
<td>8 * M bits</td>
</tr>
</tbody>
</table>

- The Address field identifies a secondary station connection address.
- The Control field specifies the function of the particular frame.
- The optional Information field contains the information data.

The IrLAP frame is also preceded and succeeded by fields that constitute a wrapping layer. The wrapping layer implements a physical layer scheme to reliably transmit the payload data. The format of the wrapper fields will vary according to the particular physical layer scheme used, but will typically include the following:

- A start flag, BOF, or STA that marks the beginning of the frame
- A frame check sequence field that allows the receiving station to check the transmission accuracy of the frame.
- A stop flag, EOF, or STO that signals the end of the frame.

Three different wrappers, depending on the speed of the connection, are described in [1].

2.1.3 IrLMP – IrDA Link Management Protocol

LMP is an abbreviation for Link Management Protocol, which implies the functionality of the protocol – to manage the link provided by IrLAP. The protocol provides support for multiple software applications/entities to operate independently and concurrently, sharing the single link provided by IrLAP [2]. That is, IrLMP provides multiple IrLMP connections on a single IrLAP connection.

Link Management in the scope of the IrDA Link Management protocol involves several things: discovery of other IrDA devices, multiplexing the link, and controlling the link.

The discovery of other entities entails each IrDA device to maintain an information base of capabilities and services that the device has available. This information can be queried from other devices.

Multiplexing the link enables multiple logical connections to exchange data over a single IrLAP link.
2.1.3.1 Link Management Components

Figure 5 is taken from [2] and shows the overall IrDA Protocol Architecture. Link Management defines two components within this architecture: the Link Management Information Access Service (LM-IAS) and the Link Management Multiplexer (LM-MUX).

As seen in figure 5, the before mentioned IAS is included as one of the Link Management components, although called LM-IAS here. This comes from the fact that IAS is a Link Management service, and therefore can be seen as a logical part of IrLMP. IAS is also defined as a part of the IrLMP specification [2].

2.1.3.2 IAS – Information Access Service

Each IrDA device maintains an Information Access Service (IAS). The IAS can be seen as a “yellow pages” describing the services available on a device.

The information in the IAS is held as a number of objects in the information base. There is a simple protocol for one IrDA device to access the information (the objects) remotely on another device - The Information Access Protocol (IAP). The IAP is the means by which two IAS entities communicate. This brings a client-server architecture where the client makes inquiries about services on a remote device and the server responds to these inquiries. The server is always found on the same location (address) within any IrDA device, at Logical Service Access Point (LSAP – see next section) zero. IAP uses the reliable data transfer provided by IrLAP.

2.1.3.3 LM-MUX

LM-MUX provides multiplexing of the IrLAP layer. IrLAP provides a reliable connection between a pair of IrDA devices. The LM-MUX provides multiple data links – multiple channels - over IrLAP.

The services provided by LM-MUX to higher layers, are accessed at logical entities called Logical Service Access Points (LSAPs). Each LSAP is identified by a simple one byte number LSAP-SEL. Services are not assigned fixed LSAP-SELS, instead services are assigned fixed names and the IAS are used to look up the LSAP-SEL of a desired service. The exception is the IAS server itself which always have LSAP-SEL zero.

Peer LM-MUX clients are connected by an LSAP-Connection. LSAP-connections between stations are carried over IrLAP connections [2]. Figure 6 illustrates the relationships between LSAP-Connection Endpoints, LSAP-Connections, IrLAP-Connections, IrLAP-Connection Endpoints and Stations.
2.1.3.4 IrLMP Services

The most important IrLMP-services include the following:

- **Device Discovery**: Uses the mechanism provided by IrLAP so that applications can find out what other IrDA devices are reachable.
- **Connect**: Uses IAS to identify an LSAP for a remote transport entity, and bounds it to the local transport entity.
- **Send Data**: Three types of data services is provided by LMP: I-Data, UI-Data and Connectionless UI-Data. I-Data uses the IrLAP reliable data transfer service to send data over a LSAP connection, and is therefore reliable in the absence of failures in the link level connection. UI-Data is sent unreliable with the IrLAP unreliable data service. An LSAP-connection is however established before the data is sent. The Connectionless UI-Data is sent unreliably without the establishment of a LSAP connection.
- **Disconnect**: Closes this LSAP-connection. Note that this does not close the underlying IrLAP connection, if not the LSAP-Connection is the only one using the IrLAP connection.

2.1.3.5 IrLMP Frame Format

All IrLMP PDU frames are sent as IrLAP data frames. IrLMP uses a two-byte header within the IrLAP frame encoded as follows:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>DLSAP-SEL</td>
<td>r</td>
<td>SLSAP-SEL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **C**: Control bit. Distinguishes between control and data frame.
- **r**: Reserved.
- **DLSAP-SEL**: The LSAP-SEL of the destination.

---

Figure 6. LSAP-Connections, IrLAP Connections and Stations.
• SLSAP-SEL: The LSAP-SEL for the sender.

When the control bit is set to 1, it indicates that the frame is a command frame, otherwise the frame is treated as a data frame. Data frames are encoded as follows:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DLSAP-SEL</td>
<td>0</td>
<td>SLSAP-SEL</td>
<td>Data</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Link control frames are encoded as follows:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DLSAP-SEL</td>
<td>r</td>
<td>SLSAP-SEL</td>
<td>A</td>
<td>opcode</td>
<td>Parameters</td>
<td></td>
</tr>
</tbody>
</table>

Where the A bit indicates whether it is a command request or a command indication. The opcode defines the type of command (e.g. Connect, Disconnect).

### 2.2 Optional IrDA Data Protocols

Of the optional protocols only IrCOMM and IrTTP was implemented in this thesis. By this, IrLAN and OBEX will not be described in detail in this section. However as short description of the two would be as follows:

• IrOBEX: The Object Exchange protocol. Easy transfer of files and other data objects.
• IrLAN: Local Area Network access, enables walk-up IR LAN access for laptops and other services.

#### 2.2.1 IrTTP – IrDA Tiny Transport Protocol

IrTTP provides per channel flow control on the IrLMP connections. Remember that IrLAP provides reliable data transfer, but this is for the IrLAP link. The introduction of multiplexed channels above IrLAP introduces a problem. As formulated by the IrTTP specification [3]:

"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 dependent 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."

The conclusion made is that the reliable service provided by IrLAP is reduced to a best-effort delivery service provided by IrLMP.

IrTTP also provides SAR (Segmentation And Reassembly). The SAR function breaks large data from higher layers into pieces, and the receiving side puts them together again (reassemble). The segmented and reassembled piece of data is called an SDU (Service Data Unit), and the maximum SDU size is negotiated when the TTP/LMP connection is established.
2.2.1.1 IrTTP Flow Control

TTP flow control is based on a “credit-giving” scheme. The receiving side gives the sender an amount of credits, which allows the sender to send as many maximum sized packets as received credits. In detail it works as follows:

- At connection establishment, each side extends some credit to the other side. Each sent credit obliges the sender to accept a maximum sized packet. The amount of credits sent is in proportion to the buffer space available. A maximum of 127 credits can be sent.
- Each packet sent causes one credit to be used up.
- The receiver is to send more credits periodically (if not congested), but the pace is entirely at the receiver’s discretion. It is however in the receiver’s interest to keep the credits flooding. If the sender is constantly running out of credits and have to wait, throughput will suffer.
- Credit-only packets are not subject to flow control, that is, credit-packets can always be sent regardless of the amount of credits in store.

2.2.1.2 IrTTP Frame Format

The two frame formats used by IrTTP are the connect-packet and the data-packet. The connect-packet is carried with the IrLMP connect packet and have therefore limited length. The format is as follows:

```
1 byte 0 to 59 bytes

| P | Initial Credit | User Data |
|
```

P = Parameter bit: 0 – No Parameters included 1 – Parameters included

The data packets are carried with the IrLMP data packets. The format is as follows:

```
1 byte IrLAP max –3 bytes

| M | Delta Credit | User Data |
|
```

M = Parameter bit: 0 – No Parameters included 1 – Parameters included

2.2.2 IrCOMM – IrDA Serial and Parallel Port emulation

IrCOMM emulates serial and parallel ports over the IrLMP/IrLAP protocol stack. By doing this, many applications (e.g. printing applications) which uses the standard APIs for serial and parallel communication can run over an IrDA link without the need of modification.

IrCOMM emulates (as of when this is written) RS-232 serial ports and Centronics parallel ports. However applications use communication ports in a variety of ways. To address this need, IrCOMM provides four service types or classes: 3-Wire Raw, 3-Wire, 9-Wire, and Centronics.
Centronics emulates parallel ports only, and is therefore not described in any further detail in this thesis. The other three service classes are described shortly in the following sections.

### 2.2.2.1 3-Wire Raw

The name of this service comes from the notion of emulating the minimum three RS-232 circuits required for full duplex communications: Signal Common, Transmitted Data (TD) and Received Data (RD).

The service type is appropriate if only the data circuits needs to be emulated (no non-data circuits like control) and when a single exclusive connection is acceptable – all other connections must be closed (except IAS-connections). This is because 3-Wire Raw uses the IrLAP flow control only (not the IrTTP flow control), which can result in deadlock conditions if other connections are open.

Because serial and parallel communication is equivalent if the non-data circuits are removed – just streams of data - it can be used to emulate both serial and parallel ports. However, no control channel means that information like flow control setting is unavailable, which makes the link merely a raw channel for the movement of data.

### 2.2.2.2 3-Wire

Like 3Wire Raw, the name of this service comes from the minimum three RS-232 circuits required for full duplex communications. However there are some major distinctions from the 3-Wire Raw service:

- 3-Wire make use of the IrTTP flow control. This brings also that 3-Wire Raw may coexist with other connections that employ IrTTP flow control. That is, it is not limited to single IrLMP connection.
- 3-Wire supports a control channel, which can send information like flow control settings etc.

### 2.2.2.3 9-Wire

From its name it is easy to conclude that the 9-Wire service emulates all the 9 circuits that are part of an RS-232 circuit. Besides the three circuits described which is emulated in the 3-Wire service classes (SD, TD and RD) it emulates the following:

- Request to Send (RTS)
- Clear to Send (CTS)
- Data Set Ready (DSR)
- Data Terminal Ready (DTR)
- Carrier Detect (CD)
- Ring Indicator (RI)

The precise description of these circuits is not in the scope of this document.

Like 3-Wire, 9-Wire uses the IrTTP flow control mechanism. It also uses the same control channel mechanism for sending control information. The control channel is used to send the states of the other RS-232 circuits as they change.

### 2.2.2.4 IrCOMM Service Interface
IrCOMM is often used together with a “Port Emulation Entity” which is supposed to emulate an existing communication API. Communication APIs vary from operating system to operating system. IrCOMM focuses on providing a set of services that can be used by all Port Emulation Entities. Figure 7 below, together with the description of its elements, is taken from the IrCOMM specification [4], and shows a model how IrCOMM fits in a typical system. Note that the IrCOMM specification does not specify how IrCOMM is used by the Port Emulation Entity. It is the job of the implementor to properly map the services of IrCOMM to the particular system.

Figure 7. IrCOMM in a typical system.

Legacy Application Applications which utilize conventional serial/parallel port communication interface.

Port Emulation Entity The port emulation entity maps a system specific communication interface (API) to the IrCOMM services. It is also responsible for device discovery and LM_LAS queries. The port emulation entity plus IrCOMM make up a port driver.

IrCOMM Provides a transparent data stream channel and control channel over an IrLMP link or Tiny TP.

Tiny TP Provides a data stream channel to IrCOMM along with a flow control mechanism.

IrLMP, IrLAP Link protocols defined by IrDA.

Port Interface Application programmer’s interface (API) for communication. This interface varies from system to system.

IrCOMM service interface The IrCOMM interface provides the following services:

- Connect
- Disconnect
- Data
- Control

2.2.2.5 IrCOMM Frame Formats

The IrCOMM Frame format depends on the service provided. There are two frame formats used in IrCOMM. The first is a very simple format used by 3-Wire raw, which contains nothing but user data. Recall that 3-Wire raw is only a data channel. The other format is used by all the other service classes, which provides a control
channel in addition to the data channel. These frames carry additional information besides the user data, creating an effect second channel, called the control channel.

IrCOMM frames fit directly into the UserData field of IrLMP or IrTTP packets. The IrLMP or IrTTP packets are based on the packet size negotiated by IrLAP (IrLAPmax). The 3-Wire raw frame format carries nothing but user data as follows:

```
| IrLAP max –2 bytes | User Data |
```

The other service classes which provides a control channel has a more elaborate frame format as follows:

```
| 1 byte | IrLAP max –3 bytes |
```

| IrTTP | Control | User Data |

The byte used by IrTTP carries credits to the other side and is not really part of the IrCOMM frame format. The control element consists of a control length byte and the control data. The control length byte holds the length of the control data. All remaining bytes are user data.

The control channel has two uses: to transmit the state of the non-data circuits in RS-232 and Centronics, and to transmit setup and status information between devices.
3 Implementation of the IrDA Data Protocols

3.1 General

The Solaris IrDA Data protocol stack is implemented as a minimal implementation; meaning that only the services needed to satisfy the communication with the CF-888 mobile phones is implemented. This also brings that only the primary station services are supported as the workstation always has the primary role in the communication with the CF-888. Further, because of the desire from Ericsson to get the Naute test system to function as soon as possible, simple fast solutions was prioritized over perhaps better but more time consuming solutions.

The protocol stack was implemented in user space. This fact comes from the “easiest and fastest solution philosophy” mentioned above, plus the fact that an implementation in kernel space would have required root access rights. This is something which perhaps sounds like something rather easy to obtain, but as the development was done on a workstation connected to the internal network it would have called for an request to the internal system administration, which is something that should be avoided at any price.

The implementation is written in C. This decision comes from the fact that the Naute test system is written in C, and also ITEX' feature to generate C code from TTCN test cases.

3.2 System design idea

The IrDA Data protocol stack is to run on top of the serial protocol or in other words, the IrDA data frames is to be sent as pure data over a serial link. This means that the serial link is still to be used as underlying communication medium and that the physical setup shown in figure 3 still is valid as a representative of the system. Figure 8 shows how the Naute system is supposed to communicate with the CF-888 phone. Notice that the IrDA frames is sent on top of the serial link and that this

Figure 8. Naute—CF-888 communication.

implies that the serial cable still can be used as medium for the communication.
As shown in figure 9, Naute sends AT-commands to the IrDA stack. The IrDA stack processes them and uses the serial port to send the IrDA frames. The phone in turn passes the incoming IrDA frames from the serial port to its internal IrDA stack, which processes them and delivers the AT-commands to the phone-processor.

One could say that the physical layer in a “normal” IrDA system, which would represent an infrared eye, is to be substituted for a serial port. Further, the stack must be integrated in the Naute test system in such a way that the AT-commands are captured and processed by the IrDA protocol stack before sent on the serial link. This brings the system overview seen in figure 9.

![Figure 9. Overview of the Naute test system with the IrDA protocol stack integrated.](image)

One thing to notice in figure 9 is the fact that each server running has its own instance of the IrDA protocol stack. This design decision derives from the “easiest solution” philosophy described before. A multi-user stack is more complex to implement due to the problem with critical areas. Examples of critical areas are buffers or queues, which are accessed by concurrent running threads. The drawback of this decision is that more memory and CPU usage is consumed. However the increased usage of both memory and CPU resources was at the moment estimated to be relatively small, and considered to be of less relevance than the aspect of making a fast and simple solution.

In order to integrate the IrDA stack as in figure 9, it is crucial to identify the exact point in the Naute test system where the integration if the IrDA stack should be performed. In other words: Find the point that handles the serial port communication and sends/receives AT-commands. That part is as discussed in the next section, the “modem” program.

### 3.2.1 Integration Idea

As mentioned before, the part of the Naute system that handles the communication to the AT-compatible modems is the “modem” communications program. The “modem” program is a simple communications program that handles a serial port and sends and receives AT commands over it. By observing this, it is easy to see that the “modem” program is the critical part of the Naute system that must be modified in order to capture the AT commands and process them with the IrDA stack before sent on the serial link. Further, the “modem” program explicitly points out what services is needed in a “Port Emulation Entity” to fully support the communication. That is, there is no point in supporting services in the Port Emulation Entity that the Naute system not uses. Further, by implementing these required services only, the implementation will also become minimal, which is a desired property.

The “modem” program handles a serial port in user space. This means that it uses the serial port API on the Sun Solaris platform. The services it uses are:

- OPEN: Open the serial port.
• CONFIGURE: Configure the serial port with desired speed, parity etc.
• WRITE DATA: Write data to the port.
• READ DATA: Read data from the port.
• CLOSE: Close the port.

By providing these services with similar “virtual” services by the means of the “Port Emulation Entity”, the “modem” program will be modified in a minimum way. However, one thing must be considered: the IrDA stack will run on top of a serial link. I.e. the physical layer will consist of a serial port and not an infrared eye, which would be the normal case. The question that arises is: what services are to be emulated by the IrCOMM-Port Emulation Entity, and what services are to be provided by the physical layer?

By considering a “normal” system, say a printer application that uses a serial port API but is to run over an infrared link, we see that this situation is different. Here the IrCOMM-Port Emulation Entity has an obvious task: to emulate the serial port API so that the printer application can run oblivious to the fact that the communication in fact is over an infrared link. In this thesis the IrCOMM-Port Emulation task is ambiguous, if not to say contradictory. It is to emulate a serial port communication API which services are supplied by the Physical layer as well. So, why not use the serial port API directly?

The answer is of course that IrDA stack must process the data before it is sent with the serial port. The emulation is actually not needed in contradiction to the Physical layer; it just makes the modification of the “modem” program nicer. The “modem” program uses Port Emulation API to do its tasks and the IrDA stack will process the data and use the serial port as a transmission medium. Actually, the serial port should be just a “raw” medium that carries data to the other side.

The decision was to let the “Port Emulation Entity” to emulate the OPEN, WRITE, READ and CLOSE services but let the physical layer do the configuration. This follows from the above discussion. The “modem” program must be able to dynamically open the port, read from the port, write to the port and close the port. These services must therefore be emulated by the Port Emulation entity. The configuration is different. The “modem” program has no interest to dynamically configure the serial link. This functionality can therefore be set to be “hard coded” in the Physical Layer.

The Physical Layer will configure the serial link to be “raw”. This means no flow control and operation on the data is to be applied. It is just to carry data. This is something that the physical layer can do and that the higher layers do not need to know about.

To let the “Port Emulation Entity” emulate the OPEN, WRITE, READ and CLOSE services brings that the “modem” program can be modified in a “clean” way. The actual functionality still lies in the physical layer where the actual serial port is handled.

By this we have the basic integration outlined: The modem program must be mapped to the emulated services provided by the “Port Emulation Entity”, the “Port Emulation Entity” will then map these services to the IrCOMM services. The IrDA stack will process all data and the physical layer will handle the serial link communication. This brings the question of the physical layer. In this thesis the physical layer will consist of a serial link. Because the IrDA stack is to be implemented in user space the simplest method is to use the Solaris serial port API to implement the physical layer.
3.3 The IrDA Data Protocols

The implementation of the IrDA Data protocols in this thesis is implemented as specified by the IrDA standards [1,2,3,4]. Most of the “protocol-specific” implementation was ported from the Linux IrDA Project [6]. The overall goal of the Linux IrDA Project is to make an implementation of the IrDA standards specifications for the Linux kernel and the code is now included in the Linux 2.2 kernel. The code used for this thesis was taken from the Linux Kernel 2.2.10 source code [7].

The following layers were taken from the Linux IrDA Project and ported to Sun Solaris in this thesis:

- IrLAP
- IrLMP
- IrTTP
- IrCOMM

The reason that the IrCOMM and IrTTP layer was implemented although they are optional is because the CF-888 has support for the IrCOMM layer and refuses to communicate with any entities that do not. Once the IrCOMM was required, so was IrTTP that provides IrCOMM with flow-control.

The fact that IAS is not mentioned here is because it is seen as a part of the IrLMP layer.

Notice that the Physical layer and the Port Emulation Entity are not ported from the Linux implementation. This is because the difference between a Solaris implementation and the Linux implementation of these two layers would be too high for a porting to be profitable. These two layers had to be formed to adapt the stack to the Solaris platform and make an integration into the Naute Test system feasible. By this, the Physical layer and the Port Emulation layer are uniquely shaped to suit the needs in the thesis.

The other layers –IrLAP, IrLMP, IrTTP and IrCOMM – follows the IrDA specifications [1,2,3,4] and contains no or little platform dependent functionality and a porting would thereby become beneficial. However, some areas in the porting from Linux to Solaris had to be considered, mainly because of the fact that the Linux protocol stack is implemented in kernel space and the Solaris protocol stack in user space. These porting specific problems are described in section 3.3.11 - Special Porting Difficulties.

3.3.1 Overall Protocol Stack Module Design

The protocol specific modularization follows from the Linux implementation. Some areas differ from the Linux implementation, as the Physical Layer and the “Port Emulation Entity”. The main difference is however the use of a Control Module in the Solaris implementation. Besides this, the overall modularization design is pretty much straightforward and follows the stack and layer view of the protocols. The overall module design is shown in figure 10.

The IrDA stack will run with only one thread of control. This means that this thread must check for the different kinds of events that can arise and take appropriate actions. This is the purpose of the Control Module. It will check for incoming events and delegate them between the other modules. The incoming events can be of three kinds: incoming data from the serial link (phone), incoming data from the user (application) and expired timers (timeouts).
3.3.1.1 Incoming data on the serial link

When there is incoming data from the phone on the serial link, the Control Module feeds it to the physical layer for processing. The physical layer will unwrap it using the wrapper module and in turn feed it to the IrLAP layer.

3.3.1.2 Incoming data from user

To understand what happens when the user sends data to be processed by the IrDA stack it is crucial to understand how the user-thread and the IrDA-thread communicates. As mentioned before, the aim with the Port Emulation Module is to emulate the serial port API. By this it has, like the serial port API, the services OPEN, READ, WRITE and CLOSE. When using the serial port API service OPEN, a filedescriptor is returned as representative for the opened port. The filedescriptor is then used as argument when the other services are applied. To emulate this behavior, the Port Emulation OPEN service will also return a filedescriptor to the user. This filedescriptor can be seen as a virtual port. Its behavior is similar to the use of an authentic port, in that it is used as argument in calls to the other services.

The virtual port is actually a socket, used as a “medium” between the user-thread and the IrDA-thread. What happens is this: When the user invokes the OPEN service it will create a subprocess and a bi-directional socket (the virtual port). The subprocess is the IrDA thread of control that will run the IrDA stack. The user thread will return with the socket as the virtual port. The socket is then used to send data between the user and the IrDA stack.

So, when the WRITE service is invoked, the user is oblivious to the fact that he really sends data over a socket. From the perspective of the IrDA stack, the Control Module will capture this data and feed it to the IrCOMM layer.

3.3.1.3 Expired Timers

The IrDA stack is pretty much time-event driven. All future events will be marked with an expiration time and queued in hashtables. Each layer possesses a hashtable where such time-events are kept. It is the Control Module’s task to scan those tables for expired timers and invoke the appropriate actions. The timer topic will be discussed further in section 3.3.11.4.
From the above discussion, the overall design of the Control Module is outlined. It can be seen as a multiplexing function that waits for incoming data or timeouts to occur. If events do occur, it takes the appropriate actions by using the different layers in the stack. If no data or no timeouts occur, it will sleep and thereby inactivate the IrDA stack until next event occurs. By this, the load on the system caused by the IrDA stack is kept on a relatively low level.

3.3.2 Physical Layer Implementation

As mentioned before, the physical layer is implemented in user-space, and handles the serial link by using the serial port API on the Sun Solaris platform. The most important services it provides include:

- **READ**: The Control Module has discovered incoming data on the serial link and uses the service to read the data. The service will unwrap the data, check it for consistence by using the services provided by the Wrapper Module and feed it to the IrLAP layer.
- **TRANSMIT**: The IrLAP layer uses this service to send data on the serial link. The service wraps the data by using the Wrapper Module and sends it over the serial link as pure data.
- **OPEN**: Will open the serial port and configure it to be in “raw” mode.
- **CLOSE**: Shuts down the serial link.

Notice that the Physical layer offers no configuration service. The configuration of the serial link is instead “hard-coded”. This is because the serial link should be a “raw” link that is to just carry data. No dynamic configuration of the link is therefore needed.
3.3.3 IrLAP Implementation

The IrLAP layer was the first layer implemented together with the most essential Physical layer functions. This was because it was crucial to see if the phone responded at all before implementing the rest of the stack. By implementing the Device Discovery service and let it search on a serial port connected to a CF-888 phone it would be possible to see if the phone responded. That the phone actually responded was considered as a break through in the thesis, as it was then proven that the theory would work. That is, the theory that the phone will pass incoming unknown data on the serial link to the IrDA stack and that it is possible to control the phone in that way over a serial link.

As the IrLAP layer was completed, the rest of the stack was implemented in a top-down manner. By implementing the stack in a top-down manner, the stack was kept as minimal as required. The services needed in the Port Emulation Entity was identified, and then the services required in the IrCOMM layer needed to support the higher layer services, and so on.

The design for the IrLAP layer can be seen in figure 11. The Finite State Machine (FSM) follows the actions and state transitions specified in the IrLAP specification [1]. The service to upper layers is accessed through the IrLAP Service Interface.

![IrLAP Design](image)

*Figure 11. IrLAP Design. Data flows, function calls.*

The Frame Wrapper is implemented as the ASYNC Wrapper specified in [1]. It collects data until an entire frame is gathered and computes the CRC for data consistency before the frame is delivered to the Frame Handler. The Frame Handler strips or adds the IrLAP header depending on if the frame is going up or down in the stack, and handles the frames by controlling its contents.

IrLAP handles a number of timers, which is controlled by the FSM. All running timers are kept in a hashtable, which makes the scanning for expired timers fast. This design is common for all IrDA layers within this thesis.
3.3.4 IrLMP Implementation

The IrLMP layer perhaps the most complex layer. It has two FSMs one for the LSAP-Connection Control and one for the IrLAP Connection Control. Because there can be multiple IrLAP connections and further multiple LSAP connections over each IrLAP connection, the different IrLAP and LSAP connections must be kept in tables that are mapped against each other. The IrLMP design is shown in figure 12.

![IrLMP Design](image)

*Figure 12. IrLMP Design. Data flows, function calls. LAP and LSAP data structure.*

In this thesis only one IrLAP connection can exist at a time. Multiple LSAP connections may however exist over that IrLAP connection. Each LSAP and LAP connection has its own state that is determined from the corresponding FSMs.

3.3.5 IAS Implementation

Although IAS is part of IrLMP it can be viewed as a stand-alone module. Figure 13 below shows its internal design in the thesis implementation.
The Information Base keeps the information as a number of objects. Each object in the Information Base has a class name for the object and a number of attributes. Figure 13 shows an example of this with an object of class “Device” which has values for two attributes - the “DeviceName” and the “IrDASupport” attributes. The Information Base has operations, which supports the registration of information, and the different layers uses these operations to register their services. For example, the IrLMP layer registers the support for IrLMP, the version of the IrLMP running and which optional LM-MUX features that are supported.

The Information Access Protocol (IAP) constitutes the service interface to access the information held in the information database. Depending on the request, this information can be held in the internal information base or at a remote host. The IAP must have support to answer these requests as well. An example of a service primitive is the GetValueByClass service which accesses all values of a named attribute in objects of a given class name. In figure 13 it shows that IAP have its own pointer to the information table, which brings that IAP can access the information without the need to go through the Information Base. The transport layers uses IAP to access information on remote devices or on the device itself.

IAP uses two FMSs depending on if it initiates or processes requests. The client machine initiates requests to the server, which processes them. These two machines are independent of one another. The client can handle an information request from a higher layer while the server is replying to a request from another client. IAP uses the IrLMP LSAP service to establish the client-server connections.

3.3.6 IrTTP Implementation

In this thesis the SAR service is not implemented. The only data sent is the AT-commands, which is in the amount of 10-50 characters (bytes) per command. So,
each AT-command fits easily in an IrLMP frame and the SAR service is in that not needed.

The flow control is implemented though. The IrTTP implementation and design is relatively simple compared to the other layers. Figure 14 shows the design. IrTTP is built up around two buffers: the receive buffer and the send buffer. Frames from higher layers are queued in the send buffer and sent when credit exists. The frames in the receive buffer is queued when delivered from the IrLMP layer, and delivered to higher layers when opportunity is given. The send and deliver operations are triggered whenever a new frame is queued, or else if no frames are queued in a given timespan the TODO-TIMER expires. The TODO-TIMER is IrTTP's only timer and makes sure that frames are sent and delivered with regular intervals.

The Credit Handler checks the receive buffer to decide whether to send more credits to the sender. More bufferspace means that more credits can be sent.

If credits exist (i.e. credits received from peer >0), frames can be sent from the send buffer. One frame for each credit can be sent.

The TODO-TIMER timeouts with regular intervals and runs the buffers. That is, tries to send buffers from the send buffer, send more credits to peer and deliver frames to higher layers.

3.3.7 IrCOMM Implementation

The IrCOMM service implemented in this thesis is the 9-Wire service. To see why this service was chosen the requirements for the Naute-Cf-888 communication must be examined.

The communication is serial which excludes the Centronics service. Left are the 3-Wire Raw, the 3-Wire and the 9-Wire services. The 3-Wire Raw service can not coexist with other (non-IAS) connections. This is because 3-Wire Raw not uses the
IrTTP flow control and relies on the IrLAP flow control solely. However, as mentioned before: each Naute-CF-888 communication link will have its own IrDA stack. In this perspective the 3-Wire Raw service would be sufficient. But, the flow control would improve by using the IrTTP flow control as well, and because the IrTTP layer is relatively simple to implement the 3-Wire Raw service was excluded.

When it comes to the choice between the 3-Wire and the 9-Wire, 3-Wire would be sufficient. The 9-Wire emulates all the 9 circuits of an RS-232 interface. By examining the “modem” communication program, which handles the original RS-232 communication to the CF-888, it shows that the extra circuits emulated by the 9-Wire service are unused. The 9-Wire service was selected anyway because of the fact that the 9-Wire service is implemented (as of when this is written) in the Linux implementation and the 3-Wire not. By the fastest and simplest solution philosophy follows that it is easier to port an existing service and just leave the unnecessary circuits unused.

Figure 15 shows the IrCOMM design for this thesis.
### 3.3.8 Port Emulation Implementation

The purpose of a Port Emulation Entity is to emulate the communication API on the platform used. In this thesis the communication API is the Solaris serial port API. As mentioned before, the services required in this thesis are the OPEN, READ, WRITE and CLOSE services.

In order to emulate the Solaris serial port API the, OPEN service returns a virtual port to the user. This virtual port is to be used as an argument with the other services: READ, WRITE and CLOSE.

The OPEN service is important in that it creates the subprocess that will run the IrDA stack. The subprocess will initiate the stack, run the “Device Discovery” service as to see if any devices are present at the link and in that case establish a connection to the device and wait for user data to send. This “Connect Establishment” procedure may of course fail. In this case the user must be notified. The OPEN service emulates the original service in that it returns a negative integer if failed. Figure 16 shows the design idea with the user thread and the IrDA thread and the bi-directional socket with which they communicate.

![Design diagram](image)

*Figure 16. Port Emulation design. Process view. Data flow and function calls.*

The Service Mapper maps the Port Emulation services to the IrCOMM services. The IrDA thread must read and write data to user the socket. Data passed from the user with the WRITE service must be read by the IrDA thread and passed to the IrCOMM layer. Data from the IrCOMM layer must be sent on the socket to the user, which can read it with the READ service. This functionality is actually part of the Control Module, which can be seen as the heart of the IrDA stack. The Control Module implementation is described next.
3.3.9 Control Module Implementation

As the entire IrDA is handled by one thread only in this thesis, some problems arise. The implementation must handle two data connections, one to the user and one over the serial link to the phone, at the same time as it must handle timeouts. This is the purpose of the Control Module. The Control Module can be viewed part as a switch that redirects data and part as a timeout handler. Incoming data from the user must be fed to the top of the IrDA stack (Port Emulation), and incoming data over the serial link must be fed to the bottom layer of the IrDA stack (Physical layer). This implies that the thread must listen to both the connections for incoming data at once.

All running timers have pointer to their timeout handlers. The Control Module must scan all running timers for timeouts and in that case run the corresponding timeout handlers. The scanning is easy in that each layer has their running timers collected in a hashtable.

As described above it looks like the thread of control must be at three times at once, or constantly checking for data or timeouts. The later would bring a high system load. The problem was solved with the `select` system call. This function has means to check multiple filedescriptors at the same time. The strength with the function is that it puts the thread to sleep until there is incoming data on one of the given filedescriptors. If no data occurs the function will block with a given timevalue. This is how the timeout problem was solved. Each execution cycle all running timers are scanned and the `select` system call is called with the timevalue of the closest pending timeout. If incoming data occurs it is read and passed to the stack, or else the thread will sleep until the next timeout when it will give control to the timeout handler in question. Figure 17 shows the idea of the Control Module.

![Figure 17. The Control Module design. The thread of control.](image)

The thread of control will handle the events on a first come first serve basis. Figure 17 illustrates how the thread distributes the control depending on the current event.
3.3.10 Integration

Thanks to the Port Emulation Entity the integration of the IrDA stack was relatively simple. Instead of the Solaris serial port API functions, the modem program was modified to call the corresponding Port Emulation functions instead. The exception is the configuration of the serial link, which was brought to the Physical layer.

3.3.11 Special Porting Difficulties

The Linux implementation is written in C, which together with the fact that Linux is built with influences from Unix makes it a logical source code to port from. There is however one major distinction between the Linux implementation and the Sun Solaris implementation made in this thesis: the Linux implementation is implemented in kernel space and the Sun Solaris in user space. This fact introduces five main areas where the code had to be totally reconstructed for the Sun Solaris implementation:

- Memory handling
- Critical areas
- The mapping between IrLAP and the physical layer
- Timer handling

These four topics will be described in the following sections.

3.3.11.1 Memory Handling

The Linux IrDA protocol stack is implemented as a collection of kernel modules and therefore uses the kernel routines for memory allocation. These memory allocation routines is replaced by user space routines for memory allocation in the Sun Solaris implementation. Two areas that are directly affected by this fact, are the frame handling and the buffer handling.

The Linux implementation uses a data structure defined in the kernel to represent frames. The operations on this data structure are well defined and include queuing-mechanisms, which makes the structure ideal for buffer handling as well. Examples are the sliding window buffer, i.e. the buffer that keeps unacknowledged frames in store in case of retransmission, and the buffer for outgoing frames to send. Both are part of the IrLAP implementation. Because buffers can be shared, the structure is also a subject for the critical areas topic discussed later.

However this code could not be ported to Sun Solaris because large parts of the operations were written in assembler. So for this thesis this data structure had to be redefined and its operation rewritten to support the representation of frames as well as its operations to support buffering, searching and so on.

Worth mentioned in this relation as well, is the data structure used for all queue representation in the Linux implementation that is a hashbin structure. As with the structure for frames and buffers this data structure had to be reworked as well. However the idea to represent queues as hashbin was kept.

3.3.11.2 Critical Areas

The critical areas topic was mentioned before in section 3.2 where figure 7 showed that each server running that handles a connection to a modem has its own IrDA stack running. This decision was founded on the purpose to make the stack free from critical areas. For example, the operations in the Linux implementation that handles the frame-buffers mentioned in section the previous section 3.3.11.1, all has critical areas (code parts) where only one thread of control is allowed access at a time. In
buffers or queues this problem is obvious. Just imagine what happens when one
thread removes a frame from a buffer at the same time as another thread tries to read
its contents.

In the Solaris implementation there are no critical areas thanks to the fact that each
server has its own IrDA stack, and all mutual exclusion mechanisms could be
removed. Which of course was at please because most of the mutual exclusion
mechanism in the Linux implementation had kernel support.

3.3.11.3 IrLAP – Physical Layer Mapping

The Linux IrDA protocol stack is implemented in kernel space. This denotes that the
IrLAP layer is directly mapped to the device driver that represents the physical layer.
If, for example, the serial port is used, the IrLAP layer will be mapped against the
serial port driver. Now, the situation for the Solaris implementation is different. Here
the IrDA protocol stack is implemented to run on top of the serial port protocol, and
the serial link is just a raw transmission medium for the IrDA stack. Whatever frame
the IrDA stack has produced at the bottom layer of the stack, the IrLAP layer, is sent
as regular data over the serial port. So, what the serial protocol does, or how it does
it, is not of interest. We just want it to carry our IrDA frame to the other side. By
this, the mapping between the IrLAP layer and the serial port could be done in user
space. The serial port is opened and configured to be in raw mode, i.e. no flow
control and no data processing whatsoever, and the IrLAP layer is mapped against
the resulting file descriptor.

3.3.11.4 Timer Handling

Each layer in the IrDA Data Protocol Stack handles a number of timers. The timers
are started and stopped according to the state charts for the Finite State Machines
(FSMs) specified in [1,2,3,4]. If a timer timeouts before it is stopped, actions must be
taken according to the state chart of the FSM in question.

In the Linux IrDA implementation, this is handled by interrupts to the kernel and
callback functions. The timed out timer will interrupt the kernel, which will handle it
by calling the specified callback function for the timer.

The Solaris IrDA implementation also utilizes callback functions. However, because
it is implemented in user space the timeouts are not handled by interrupts. Instead,
each layer utilizes a hashtable implementation to keep track of all its running timers.
Each execution cycle, the hashtable for each layer is scanned for expired timers. If
expired timers are found, they are removed from the hashtable, and the
 corresponding callback functions are called.

3.4 Evaluation of the IrDA Implementation

The IrDA Protocol Stack was implemented on time, as scheduled. The stack was
tested during implementation by sending simple AT-commands from a terminal
window to a connected phone. The testing was extended to embrace multiple
phones. Finally the stack was integrated into the Naute test system to carry the AT
commands. The tool Purify® was used on a running basis to perform run time
memory checks.

The fact that each server instance has its own stack showed to have no or little
implications on the system performance. This is thanks to how the Control Module
was implemented with the select system call (see section 3.3.9).

The implementation shows only one major limitation. The critical part when running
the Naute Test System with the IrDA protocol stack is the connection establishment
to the phones. A connection fails to be established with an approximate rate of 1 to
20. The reason for this is that the serial driver, on the Sun SPARCstation used,
interrupts with a minimum interval of 20 milliseconds. The effect of this is that a read request on the serial port, immediately following another read request, will be blocked for at least 20 milliseconds. So, if the first read request fails to read an entire frame sent from the phone, the next part will not be read until 20 ms has elapsed. This has a fundamental impact on the protocol behaviour when 20 milliseconds is a relatively long time period in the perspective of the IrDA protocol. A possible scenario would be that the protocol thinks the link is idle, or sends re-requests for frames the phone already sent. When a connection has been established, the impact of this problem is however minor. It is in the establishment of the connection that problems occur. The chance that a connection fails is of course dependent on how many phones it involves. More phones mean that the chance that one fails to connect is higher. At the moment the testing with the tool is in small scale and often only includes two phones. In this case the problem is minor. If the testing is to include up to fifteen phones the problem is impossible to avoid. It is however important to consider the fact that the testing is only an evaluation on the use of testing tools. No full scale testing with the Naute system is planned, and the solution with the IrDA stack implementation is only a temporary solution until the new version of the CF-888 comes. Therefore it is possible to say that the implementation works well in this phase of the evaluation. Up to three phones can be included in the test cases without any higher problems, and this is sufficient to run a large quantity of different test scenarios.
4 TTCN Test Case Implementation

A Test Step Library has been created in the Naute tool ITEX. The intention with this library is to form the basic building blocks for future generation of test cases. The test steps are grouped in sub-directories according to their intended context. Each group of test steps is described shortly in the following sections.

4.1 Test Step Library

Before the different groups are described it can be good to understand some of the CF-888 phone’s functionality. The phone is controlled with AT-commands. A variety of AT-commands exist which can be used to control the phone. There are for example commands for voice call handling, data call handling, fax call handling, SMS-handling, phone configuration etc. The difference between voice, data and fax call is as it sounds. Voice calls are connections established for speech. Data calls are used to send generic data, and fax calls is of course intended for fax communication. This outlines a logical grouping in itself. The Test Steps are grouped thereafter in the following sub-directories:

- INIT: Initiate and configure Test Steps.
- CONTROL: Generic phone control Test Steps.
- VOICE CALL: Test Steps to establish and handle voice calls.
- DATA CALL: Test Steps to establish data connections, send and consistency check data.
- SMS: Short Message Service (SMS) Test Steps.

The Test Steps in each group is described in the following sections.

4.1.1 INIT Group

This group contains Test Steps for initiation and configuration of the CF-888 phone.

4.1.1.1 Remove Pending Input

This Test Step removes any pending input for the specified CF-888. It could be used to flush the input buffer.

4.1.1.2 Init CF888

This is a mandatory Test Step that must be called, for all the involved phones, first in any Test Case. The Test Step performs three tasks:

- Establishes a connection to the CF888.
- Configures the phone to disable the command line echo.
- Resets the phone to user defined configuration.

4.1.2 CONTROL Group

The CONTROL_LIB is intended to contain generic control commands.
4.1.2.1 CMD Test Step

This Test Step takes an AT command as an argument and sends it to the defined phone. The desired response is also passed as an argument, and The Test Step will fail if the phone's response does not correspond to it.

4.1.3 VOICE CALL Group

This library contains test steps to handle voice calls. The phone can make three kinds of connections: voice calls, data calls and fax calls.

4.1.3.1 Voice Call Up

This test step makes a voice call between two phones. Fails if the connection can not be established.

4.1.3.2 Voice Call With Time

This test step establishes a voice call between two phones, and holds the connection for a defined period before it hangs up.

4.1.3.3 Voice Hang Up

Hangs up the current voice connection.

4.1.4 DATA CALL Group

As the above mentioned 'VOICE CALL Group', but in this case for data calls.

4.1.4.1 Data Call Up

This test step makes a data call between two phones. Fails if the connection can not be established.

4.1.4.2 Data Hang Up

Hangs up the current data connection.

4.1.4.3 Send Data

Sends defined data between two phones connected via the ‘DATA_CALL_UP’ Test Step. The sent data is checked for consistency on the receiving side, and the test step will fail if not consistent. A data stream can be emulated by calling this Test Step repeatedly with small data strings.

4.1.5 SMS Group

Handles SMS-services, as sending SMS messages, storing SMS messages in memory etc.

4.1.5.1 Set SMSC Address

Sets the address to the SMSC (Short Message Service Center) in the phone. This is necessary to inform the phone where to send SMS messages.
4.1.5.2 SMS Store
Stores a given sms message in the phone's storage.

4.1.5.3 SMS Send
Sends a sms message to the SMSC declared with the 'SET_SMSC_ADDRESS' Test Step. The receiving phone's address is contained in the sms message pdu sent.

4.2 Evaluation of the TTCN Test Step Library
The written Test Steps are intended to be used as basic building blocks for the writing of Test Cases that involves the CF-888 cell phone. The most fundamental functions are covered by the Test Steps and can be used to generate a wide variety of Test Cases. Examples are Test Cases for sending SMS messages and Test Cases for generating data connections and sending data with consistency checks. Future more complex Test Cases will of course call for the need of a more extensive Test Step library.
5 Result

All the goals of the Master Thesis have been fulfilled.

- The IrDA Data Protocols have been implemented on a Sun Solaris platform
- Full functionality when controlling the CF-888 cell phone over a serial link have been obtained
- The implemented protocol stack has been integrated into the Naute Test system so that the system can control the CF-888 cell phones according to the executed test cases.
- A Test Step Library have been written in the formal test language TTCN with the tool ITEX to form a basic foundation for future Test Case writing for the CF-888 cell phone.
- A report that describes the thesis work and documents the design solutions have been written.

The resulting test system works very well for a small number of phones. As the number of phones increases, the connection establishment becomes critical.

Whether or not Ericsson decides that the use of formal testing tools will make the testing of AXE systems more effective, is for the future to decide. So far he evaluation looks promising, and the work continues within the testing and verification department by extending the Test Step Library with more Test Steps. The use of the testing tool is of course dependent on if meaningful Test Cases can be generated, and in that case if they can be gathered to run in sequence, so that the use of human interaction is minimized. The idea is of course that the time for generating Test Cases, running them and analyze them will fall below the time for traditional testing where the testing is done totally by human interaction. This is why a good Test Step library is of importance. The Test Case generation should be fast and simple which is something that can be obtained by using a good Test Step library. All underlying complexity that is of no importance from the perspective of the tests can be put in the Test Steps.

The use of efficient testing tools is of course not just dependent on the quality of the tools but also in which context it is used. Not all testing will benefit from the use of such tools.

One thing to consider is that these kinds of test tools are relatively new on the market and the suites will become better and more efficient. The Telelogic Tau suite has already got automatic code and test generation from SDL and UML diagrams. The generated tests are of course incomplete and must be supplemented with human tests to cover all of the systems functionality. The technique exists however and will improve, and the amount of human generated test cases can be reduced if used.

5.1 Further Work

As the new version of the CF-888 comes, the use of the implemented IrDA protocol stack will be of no use. The question of whether improvements to the IrDA implementation is meaningful or not, is dependent on when the release will be. If such an improvement is decided to be meaningful, then there is really only one thing to improve and that is the connection establishment problem. The solution to this is to implement a new serial port device driver with better granularity than 20 milliseconds.
The Test Step library can of course be extended endlessly. The trick is however to form a good foundation to stand on. More Test Steps does not necessarily mean a better foundation. A small but well thought out collection of test steps is better. The goal would be to reuse as much code as possible.
References

http://www.irda.org/standards/pubs/IrData.zip

http://www.irda.org/standards/pubs/IrData.zip

http://www.irda.org/standards/pubs/IrData.zip


http://www.irda.org

http://www.cs.uit.no/linux-irda/

[7] Linux 2.2.10 Kernel Source.
http://www.kernel.org/pub/linux/kernel/v2.2/linux-2.2.10.tar.gz


http://www.telelogic.com