MASTER

System level study of a wireless tablet

Niessen, P.C.J.G.

Award date:
2003

Link to publication
System level study of a wireless tablet

Ing. P.C.J.G. Niessen

Coach: prof.dr.ir. J.L. van Meerbergen
Supervisor: prof.dr.ir. J.L. van Meerbergen
Date: 6 January 2003 – 21 November 2003
Title: System level study of a wireless tablet.  
On the extension of iPronto for streaming media.

Author(s): Niessen, P.C.J.G.

Keywords: mobile-computing; software-architecture; multimedia-computing; video-signal-processing

Abstract: The project investigates the extension of the iPronto, a portable, wireless connected tablet, with streaming media. The current iPronto design is not capable of real-time VGA sized media decoding. The report focuses on streaming media inside the device. An analysis is done on the software and hardware architecture of the system. The analysis shows that timing is hard to guarantee in both the software stack, as in the hardware platform. To extend the iPronto with streaming media, the proposal is to separate the design into a control and streaming domain. The report shows how to extend the current iPronto software stack to support streaming media.

Conclusions: A separation into two domains is proposed, a control and a streaming domain. The streaming domain contains all signal processing functionality of the system. A control domain abstraction for the streaming domain is proposed, consisting of software components that represent the signal processing functionality. A grouping concept for components, called services, is introduced. The services are meaningful building blocks for applications. It is feasible to extend the current iPronto software stack with these concepts.
Contents

1. Introduction ................................................................................................ 1
   1.1. iPronto functionality ............................................................................. 1
   1.2. Problem description ............................................................................. 2
   1.3. Overview of the report ......................................................................... 4

2. iPronto description .................................................................................... 5
   2.1. Software stack ..................................................................................... 5
   2.2. Hardware description ........................................................................... 7
   2.3. Comparison with a PC architecture ..................................................... 9
      2.3.1. Software .................................................................................... 9
      2.3.2. Hardware ................................................................................ 10

3. Analysis .................................................................................................... 12
   3.1. Control ............................................................................................... 12
      3.1.1. Java thread mapping on the OS.............................................. 12
      3.1.2. Profiling options ....................................................................... 14
      3.1.3. Measurement setup ................................................................ 16
      3.1.4. Scenarios ................................................................................ 17
      3.1.5. Measurement results ............................................................... 18
   3.2. Streaming analysis ............................................................................ 22
      3.2.1. LCD refresh data ..................................................................... 22
      3.2.2. Wireless network traffic ........................................................... 25
      3.2.3. Interference LCD and wireless data traffic .............................. 27
   3.3. Conclusions ....................................................................................... 28

4. Conceptual architecture .......................................................................... 29
   4.1. Separation streaming and control domain ......................................... 29
   4.2. Hardware architecture ....................................................................... 30
   4.3. Software architecture ......................................................................... 32
   4.4. Conclusions ....................................................................................... 39

5. Consequences for iPronto ...................................................................... 41
   5.1. Service layer ...................................................................................... 41
   5.2. Options for hardware extension ......................................................... 48
      5.2.1. Wireless network interface ...................................................... 49
5.2.2. Video processing functionality ................................................. 49

6. Conclusion and recommendations .............................................. 55
   6.1. Conclusions .............................................................................. 55
   6.2. Recommendations .................................................................... 55

References .......................................................................................... 57

A Measurement CPU load Web browser at start-up on iPronto. .......... 60
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>The iPronto device.</td>
<td>1</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Example device set-up of an Internet connected home network.</td>
<td>3</td>
</tr>
<tr>
<td>Figure 3</td>
<td>iPronto software stack.</td>
<td>5</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Detailed iPronto software stack.</td>
<td>6</td>
</tr>
<tr>
<td>Figure 5</td>
<td>OSGi bundle state diagram.</td>
<td>6</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Internal structure of the PXA255 SoC processor.</td>
<td>8</td>
</tr>
<tr>
<td>Figure 7</td>
<td>iPronto hardware set-up.</td>
<td>9</td>
</tr>
<tr>
<td>Figure 8</td>
<td>The different Java VM specifications and their target devices.</td>
<td>10</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Desktop PC architecture.</td>
<td>11</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Thread mapping options for a Java virtual machine.</td>
<td>13</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Java thread mapping to an OS process in iPronto.</td>
<td>13</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Profiling options depicted in a simplified iPronto software stack.</td>
<td>14</td>
</tr>
<tr>
<td>Figure 13</td>
<td>General set-up for profiling Java software.</td>
<td>15</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Screenshots of the applications in the remote control scenario.</td>
<td>18</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Applications used in the second scenario.</td>
<td>18</td>
</tr>
<tr>
<td>Figure 16</td>
<td>CPU load and memory usage of the boot scenario.</td>
<td>19</td>
</tr>
<tr>
<td>Figure 17</td>
<td>CPU load and memory usage of scenario 1.</td>
<td>20</td>
</tr>
<tr>
<td>Figure 18</td>
<td>CPU load and memory usage of scenario 2.</td>
<td>21</td>
</tr>
<tr>
<td>Figure 19</td>
<td>CPU load and memory usage of scenario 3.</td>
<td>22</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Internal LCD controller set-up.</td>
<td>24</td>
</tr>
<tr>
<td>Figure 21</td>
<td>LCD traffic pattern on the system bus of iPronto.</td>
<td>25</td>
</tr>
<tr>
<td>Figure 22</td>
<td>IEEE 802 Ethernet protocol stack.</td>
<td>26</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Frame format as defined by IEEE 802.11</td>
<td>26</td>
</tr>
<tr>
<td>Figure 24</td>
<td>Separation control and streaming domain.</td>
<td>29</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Task graph representing the processing pipe.</td>
<td>30</td>
</tr>
<tr>
<td>Figure 26</td>
<td>Combined background memory for streaming and control.</td>
<td>31</td>
</tr>
<tr>
<td>Figure 27</td>
<td>Template of the streaming domain hardware.</td>
<td>32</td>
</tr>
<tr>
<td>Figure 28</td>
<td>Streaming services and components positioned in the SW stack.</td>
<td>33</td>
</tr>
<tr>
<td>Figure 29</td>
<td>Example requirements graph and corresponding service graph.</td>
<td>34</td>
</tr>
<tr>
<td>Figure 30</td>
<td>Several possibilities of component / task combinations.</td>
<td>35</td>
</tr>
<tr>
<td>Figure 31</td>
<td>Placement graph and resource manager in the SW stack.</td>
<td>37</td>
</tr>
<tr>
<td>Figure 32</td>
<td>Example of a component with unconnected inputs.</td>
<td>38</td>
</tr>
<tr>
<td>Figure 33</td>
<td>Placement streaming services in the iPronto software stack.</td>
<td>41</td>
</tr>
<tr>
<td>Figure 34</td>
<td>Diagram of the main MMAPI classes and interfaces.</td>
<td>42</td>
</tr>
<tr>
<td>Figure 35</td>
<td>State machine of the player object.</td>
<td>44</td>
</tr>
<tr>
<td>Figure 36</td>
<td>Extension of the software stack for streaming media.</td>
<td>48</td>
</tr>
</tbody>
</table>
List of Tables

Table 1: Media formats to support. 4
Table 2: Available performance objects of interest. 16
Table 3: Observed performance indicators. 16
Table 4: Task processing load(%) example per processing unit. 36
Table 5: Schemes defined by MMAPI. 43
Table 6: State description of the Player object. 44
Table 7: Identified Player classes 45
Table 8: Identified services 45
Table 9: Player state transaction coupled to the logical graph. 47
Table 10: CPU load MPEG-4 software decoder for an ARM9 processor 49
Table 11: Feature summary of two Virtex-II Pro FPGA family members. 50
Table 12: Summary of the main features of several chipsets. 52

Summary of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programmer Interface</td>
</tr>
<tr>
<td>OSGi</td>
<td>Open Services Gateway Initiative</td>
</tr>
<tr>
<td>J2ME</td>
<td>Java specification for small portable devices (Micro Edition)</td>
</tr>
<tr>
<td>J2SE</td>
<td>Java specification for desktop computers (Standard Edition)</td>
</tr>
<tr>
<td>J2EE</td>
<td>Java specification for servers (Enterprise Edition)</td>
</tr>
<tr>
<td>JMF</td>
<td>Java Media Framework</td>
</tr>
</tbody>
</table>
1. Introduction

People nowadays more and more using portable devices in the home or on the move. Typical portable devices currently in use are used mobile phones and portable mp3 players. Simultaneously the amount of stationary, fixed devices in the home is also increasing. A typical household contains at least one TV, a personal computer, an audio surround set and a DVD player. Next, with the introduction of broadband Internet access networking functionality is entering the home. Stationary and portable devices in the home will become more and more connected to each other via the network.

One of the current products of Philips Remote Control Systems, part of Philips Consumer Electronics, is the iPronto device. The iPronto is an in-home portable display intended as easy to use centre of control in the home. We will use the iPronto as carrier for a study to extend a portable tablet with support for the playback of audio and video.

1.1. iPronto functionality

The iPronto is a mobile wireless device that combines the functionality of a remote control, an electronic program guide and a web browser. The device features a 6.4" VGA color touch screen that enables the user to use the device as the center of control in the home.

![Figure 1: The iPronto device.](image)

On the left side of the screen a soft button bar is located. Each button is connected with a specific application. By using one of the buttons the active program is switched. This enables the user to use multiple applications. The five buttons beneath the screen are used to directly control a TV. The buttons are a volume up and down, next and previous channel and a mute button. Next a small LCD screen provides icons for battery status and a clock.

The remote control application uses the touch screen to show a virtual remote control. On the screen the buttons of a remote control are drawn. On touching one of the buttons on screen, the iPronto sends out the corresponding remote control infrared code. On the left-hand side of the remote control application
the user can change between remote control layouts for devices like TV, DVD or an audio amplifier. With the PC application 'iProntoEdit' the user can customize the screen layout and use other images for the buttons and background.

The electronic program guide (EPG) provides the user to see the TV program list of various TV channels. With the EPG the user can select programs using the genre or read descriptions of the TV programs. The program information is downloaded at night from the Internet via the wireless interface and stored in a database.

The web browser application displays World Wide Web pages made in the HTML format. It includes support for JavaScript. Users can enter an Internet address by means of a virtual pop up keyboard. The virtual keyboard is also used to edit the iPronto device settings, like LCD brightness, network settings and audio feedback on user actions. Three soft buttons are linked to the web browser application, each with a different home page.

The iPronto is currently targeted at the high-end market segment. For full functionality it requires an installed wireless base station that connects to the Internet. The iPronto software can be upgraded on user request from the device settings menu. During an upgrade it downloads a new software image from the Internet and copies it into internal flash memory.

Current applications have a static nature when started, they are event driven on user input. Reaction time of the application functionality is direct, switching of applications is reasonable though noticeable. The current applications do not have any hard real-time features.

1.2. Problem description

The iPronto, described in the previous paragraphs, is a versatile device. One of the lacking features of the device is the capability to playback video. It is foreseen that the user expects the device to playback content stored on a remote device. This paragraph outlines the typical environment in which the iPronto should support playback of content from a remote device.

Use case scenarios

The playback of audiovisual material on the iPronto can be classified into two scenarios:

1. Playback from a device located in the home network.
2. Playback from a remote server on the Internet.

The typical device configuration of both scenarios is given in Figure 2. In the first scenario the iPronto uses the DVD player or the desktop PC in the home as the content source. In the second scenario the iPronto fetches the content from a server on the Internet.
This report will limit itself to streaming functionality in the iPronto device itself. The streaming functionality over the network is not addressed, though it provides boundary conditions.

**Network connectivity**
The iPronto is connected wireless to the home network. The home network can consist of a mixed wired/wireless infrastructure and is connected to Internet via a gateway. The home network is assumed open, each device in the home network can communicate freely with other equipment in the home network. Furthermore the home network supports the Internet Protocol (IP) and the accompanying transmission control protocol (TCP) and user datagram protocol (UDP). For the networking layer beneath the IP layer, the usage of the Ethernet protocol is assumed, as it currently is the dominating technology in the home networking market.

For the data transfer in the home network the hyper text transfer protocol (HTTP) or the real time transport protocol (RTP), both on top of IP, are used.

**Supported media formats**
The device should support the decoding of a wireless received video and audio stream with a maximum bitrate of 10 Mbps for an MPEG-2 encoded data and 8 Mbps for a MPEG-4 encoded data. The characteristics of both media formats are described in Table 1.
Table 1: Media formats to support.

<table>
<thead>
<tr>
<th>Max. bitrate [Mbit/sec]</th>
<th>MPEG-2</th>
<th>MPEG-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Visual) Profile</td>
<td>ML@MP</td>
<td>Advanced Simple</td>
</tr>
<tr>
<td>Level 0 till 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Picture width x height</td>
<td>720 x 576</td>
<td>720 x 576</td>
</tr>
<tr>
<td>Frames per second</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

The MPEG-2 requirements are based on the ability to decode a standard DVD movie, without using transcoding facilities (lowering bit-rate, image quality) at the source. For audio we propose to support stereo decoding only, as the iPronto is equipped with a set of stereo speakers. The maximum bitrate of 10.08 Mbit per second is based on the multiplexed data rate of DVD. This is 5 Mbps lower than the maximum bitrate of the Main Level @ Main Profile from the MPEG-2 standard. Note that the maximum bitrate exceeds the current capabilities of the 802.11b wireless network interface on the iPronto.

The MPEG-4 requirements are based on decoding a TV sized image format (NTSC, PAL). The profiles support only rectangular video. The profiles supported are Simple and Advanced Simple. These profiles support image sizes from QCIF (176 x 144 pixels) up to PAL (720 x 576 pixels), corresponding with profile levels zero till five of the Advanced Simple profile. The Simple profile is a subset of the Advanced Simple profile. The reader should be aware that the intention is to only support above mentioned profiles, which use rectangular shaped video objects. Wherever the term MPEG-4 is used in this report, the context is the MPEG-4 visual profile with rectangular shaped video.

Problem statement
Within the context described in the previous paragraph, the problem statement of the report is:

“How to effectively extend the current iPronto design, from a system point of view, to support playback of streaming video from a remote location. Taking both hardware and software aspects of the system into account.”

1.3. Overview of the report

The next chapter of the report gives an overview of the software and hardware architecture of the iPronto. Chapter 3 describes the analysis done to better understand the software interaction between on the performance of the software and hardware described in chapter 2. In chapter 4 the concepts to extend the iPronto with streaming video are introduced. How these concepts fit into the current iPronto set-up are described in chapter 5. Finally, in chapter 6 conclusions and recommendations for further work are given.
2. iPronto description

The main functionality of the iPronto is described in section 1.1. The sections in this chapter explain the software stack and the hardware architecture of the iPronto. The software stack and hardware are explained up to a basic level, to understand the other chapters in the report. No detailed information of the internals of the software or the hardware is discussed, nor necessary to understand the rest of the report.

2.1. Software stack

The software stack of the iPronto, given in Figure 3, is ordered in a stacked view of modules that each consist of a set of defined application programmer interfaces (APIs) and a clear definition of their semantics. This module view gives the static relations in the software (SW) architecture. The main SW elements are all based on the Java programming language.

![Figure 3: iPronto software stack.](image)

At the top of the software stack the applications reside. The applications make use of the available services. Both applications and services are components conforming to the Open Services Gateway initiative (OSGi) specification[1]. The OSGi specification defines a service framework that provides life cycle management for components and permits application developers to partition applications into small self-installable components. These components are called bundles in OSGi terminology.

Bundles can be downloaded on demand and removed when they are no longer needed. When a bundle is installed and activated in the framework, it can register any number of application programmer interfaces that can be used by other bundles. This dynamic aspect makes the software extensible on the device after deployment: new bundles can be installed for added features or existing bundles can be updated for bug fixes without bringing down the entire system. Deployed bundles are executed inside the OSGi framework and find a well-defined and protected execution environment.
The services and applications in Figure 4 are all OSGi bundles. Services provide functionality common to one or more applications. On the start of an application the OSGi framework resolves all references to services from the application and loads and starts service bundles if required. Next the requested application is started. Figure 5 shows the bundle states the framework uses.

Applications and services differ in the fact that application bundles are transient. Services are usually resident. Application are ‘closed’, they do not provide functionality to other bundles. Services provide functionality to other bundles and can depend on other services. Finally applications contain a user interface and services don’t provide a user interface.

In the iPronto all bundles are located in flash memory. Multiple service bundles are running simultaneously and only one application bundle is running at a time. If another application bundle is started, the current running application is stopped. The OSGi framework does support multiple running applications,
but the iPronto currently only allows one active application.

Next to application management the OSGi specification defines additional services like logging and http. These additional service definitions are not used in the iPronto environment.

The Java environment is based on the Java 2 Micro Edition (J2ME) specification\(^1\). The J2ME provides a Java virtual machine with libraries and APIs tailored towards resource constraint devices like PDAs and Web tablets. The footprint of the Java environment is in the order of 1 to 10 MB. The J2ME environment is upward compatible with the Java2 Standard Edition (J2SE), as used on desktop PCs, and the server version Java2 Enterprise Edition (J2EE).

The main reason for using Java is to ease the portability of applications over multiple product platforms. Thus the applications developed for the iPronto product should be easily transferable to newly developed products. In terms of the software stack in Figure 4 the service and application layer should as platform independent as possible.

### 2.2. Hardware description

At the heart of the iPronto design is the Intel PXA255\([2][1]\) SoC processor. The PXA255 contains an XScale RISC processor core that is ARM version5 instruction set architecture compatible. Next the XScale is extended with a 40 bit wide multiply and accumulate unit for media processing.

Next to the XScale processor core the PXA255 contains an on-board LCD controller, memory controller and DMA unit. These units are all connected to the internal 200 MHz bus. A bridge connects a large set of peripheral units of which most provide connectivity to modern connectivity protocols. The internal structure of the PXA255 is shown in Figure 6.

The memory controller interfaces to the SDRAM, Flash memory, PCMCIA/CF slots and an optional companion chip. Although Figure 6 shows four separate connections between the memory controller and the external devices, one address- and data bus is shared between all devices. The memory controller applies separate control signals with programmable timing for the different memory types. Next a companion chip can request control over the external bus.

\(^1\) J2ME is the successor of the personal Java specification
Figure 6: Internal structure of the PXA255 SoC processor.

The board level schematic of the iPronto is shown in Figure 7. In the iPronto the PXA255 is configured with 64 MB SDRAM memory and 64 MB of flash memory. The PCMCIA interface is used to connect to an IEEE802.11b wireless LAN card. The wireless LAN card provides connection to the Internet via a base-station. The on-chip LCD controller connects to a 6.4" VGA color touch screen.
2.3. Comparison with a PC architecture

In this section a comparison between a PC architecture and the iPronto will be performed. A comparison between the two architectures can be made along different criteria of the system. Properties that could be compared are for example the user interface concepts, the software architecture, hardware set-up, cost and weight. The user interface concepts will not be discussed. The next sections will compare the software architecture and the hardware architecture of both systems. Furthermore a comparison with respect to the power consumption also shows interesting insights.

2.3.1. Software

At operating system level the iPronto uses a tailored Linux kernel. The Linux kernel is tailored in the sense that it only contains hardware drivers for the on-board hardware. The memory management system only uses physical RAM, no swap space on a hard disk is available.

The use of a micro-edition Java virtual machine adds a small footprint virtual machine that is tailored to embedded devices. The virtual machine conforms to the CDC profile, which means that a limited set of Java APIs from libraries is available. The graphical user interface (GUI) system is a lightweight Java based version. The GUI system has smaller computational requirements with
respect to a desktop PC solution, but it imposes some restrictions in the construction of the GUI by the programmer.

![Diagram showing Java VM specifications and target devices.](image)

*Figure 8: The different Java VM specifications and their target devices.*

On the application level the user can only start a predefined set of applications and the system is always in control of the applications and services running. This prevents thrashing of the system as opposite to a PC where the user can start an infinite amount of applications.

The applications in the iPronto are programmed in the Java language and apply strong software decoupling methods to prevent platform dependencies. Changes in the HW platform therefore only need rework at the lower software layers, so the applications do not have to be ported. In the PC environment a big emphasis is on backwards instruction set compatibility in order to use existing applications.

### 2.3.2. Hardware

The PC architecture is focussed on general purpose computing. It uses a single fast CPU with a large (background) memory. Next a special purpose graphics adapter is used that directly connects via the high speed AGP bus to the ‘North Bridge’. The North Bridge regulates the traffic from the graphics adapter and the CPU and connects to the ‘South Bridge’ via a high-speed bus.

The South Bridge acts as the central connection point for all devices with lower bandwidth and latency requirements. The architecture of a PC is still evolving, for example the connection between the North and South Bridge developed from a PCI bus to a dedicated bus. Newer designs with integrated Gigabit Ethernet directly connect the Ethernet to the North Bridge for bandwidth reasons.
In notebooks the hardware architecture has a similar set-up but with some more component integration and lower speed components. The North Bridge can be integrated with the CPU and the South Bridge is a PCI device that connects to the PCI bus from the North Bridge.

The iPronto has a limited amount of (background) memory. A hard disk is not available. The Flash memory is used as permanent storage. It is organized into partitions, accessible via the Linux file system and is very small, 64 MB, in comparison with small PC hard disks.

The PXA255 is passively cooled, in contradiction to a PC that uses a forced cooling. Using an x86 compatible processor comes with an additional price level and much more power consumption. An important property of the iPronto design is the thinness of the product, it should be as thin as possible. Embedded processors are in general thinner than x86 compatible processors. A PC architecture is often PCI oriented with a corresponding item for the power budget. The absence of a PCI bus limits the expandability. The number of components and busses in iPronto is much smaller. Both PC and iPronto hardware architectures have in common that the memory access is the main bottleneck.

Concluding, the iPronto is tailored to low power, small form factor and low cost for a set of specific applications.
3. Analysis

To find problems or limitations of the current setup an analysis was performed regarding the execution architecture of the iPronto.

3.1. Control

The software stack of the iPronto, described in chapter 1, is a functional decomposition of the software divided into layers with subsystems and modules. The layers represent allowable interfaces between the various modules in the system. They reduce and isolate internal and external dependencies, which facilitates development and testing of the various subsystems. Modules within a layer can communicate with each other. Modules in different layers can communicate with each other if their respective layers are adjacent. In addition dependencies between layers are top down oriented, modules located in a higher layer only have dependency to lower layers, they do not depend on modules located higher in the stack.

The functional stacked software view on the system does not contain information on the run-time behavior of the system. This means that it does not contain information on the various operating system tasks or processes, communication mechanisms or resource usage. In order to obtain information on the dynamic configuration, which will enable prediction on the performance of the system, the run-time structure of the system needs to be examined.

The next section will describe the mapping of Java Threads to operating system processes. Section 3.1.2 will describe options for software based profiling of the system. Section 3.1.3 details the measurement set-up and 3.1.4 defines 'use-case' scenarios that exercise the system in the measurement set-up. The results of the measurements are given in section 3.1.5.

3.1.1. Java thread mapping on the OS

This section deals with Java threads and operating system processes. We define these terms as:

thread A thread is a single line of execution. Threads execute independent instruction flows on a shared memory. Multiple threads exhibit the behavior of concurrently executing instruction flows.

process A process is an instance of a running program and is a combination of private memory space and the resources used by the program.

Operating systems support the notion of a process[3]. A process executes one instruction flow thus has one thread by default. To support multiple execution flows operating systems support multiple threads that share the memory space of a process and each have a separate execution flow. Switching threads is far more efficient than switching processes. Therefore the notion of a thread is sometimes also called a lightweight process.
a thread is sometimes also called a lightweight process.

The Java language provides the thread concept[4]. Java threads have priority. Threads with a higher priority are executed in preference over threads with lower priority. The Java virtual machine specification does not specify how the Java threads are exactly scheduled. This is left to the virtual machine implementation. In the case of a virtual machine running on an operating system that support threads, three options can be distinguished that connect (map) Java threads to operating system (OS) threads. The thread mapping options are given in Figure 10.

The first option is a ‘one to one’ mapping of Java threads onto OS level threads. The second option maps all Java threads into a single native OS thread. The “many to many” mapping option maps multiple Java threads to multiple OS level threads, where each OS level thread can be used by multiple Java threads.

![Figure 10: Thread mapping options for a Java virtual machine.](image)

Due to the operating abstraction layer in iPronto, called elate, the mapping of Java threads to the Operating System process is somewhat more complicated. The abstraction layer is placed between the Java threads and the OS process. The Java VM runs in the elate layer and applies a one to one mapping of Java threads to intent threads. The intent threads are mapped via the M:1 model into an OS process. An illustration is given in Figure 11.

![Figure 11: Java thread mapping to an OS process in iPronto.](image)
Attention should be paid to blocking I/O calls. A blocking I/O in a Java thread could stop the execution of program code in the thread. Normally the thread scheduler in the VM, here implemented in the intent layer, can activate another thread if this occurs. But Java code that calls native (OS layer) libraries must be aware that the complete Java VM is not halted on blocking I/O call. For Java libraries included with the VM the supplier of the VM takes care of this.

Some hardware in the iPronto is not supported by the default Java configuration. As example we will take the IR receiver. As the Java specification does not include IR support, a separate developed Java library supports the usage of IR codes. The Java library uses a C library from the OS, via a Java native interface (JNI), to drive the IR hardware. To prevent blocking of the complete Java VM when the OS library is called a separate thread in the OS process is created used to circumvent the blocking problem. A corresponding OS thread is created for each Java thread that uses JNI calls to native libraries [5].

3.1.2. Profiling options

Acquiring more detailed information on the resource usage of the iPronto can be done in several ways. The first method is to collect performance data of the Java software in the iPronto. Another option would be to collect performance data on the complete Java environment, including the Java virtual machine that executes the Java bytecode. These options are reflected in Figure 12.

![Figure 12: Profiling options depicted in a simplified iPronto software stack.](image)

The first option in Figure 12 is to profile the Java software with a Java virtual machine profiler interface. This profiles the software above the Java virtual machine layer. The second option is to gather information using the OS layer, thus the Java software including the virtual machine. The Java virtual machine from TAO Corporation is named “intent JTE”. The next paragraphs will explain the profiling options with more detail.

**Java software profiling**

The Java virtual machine specification defines interfaces for debugging and profiling. These interfaces are optional and allow external programs to connect to the virtual machine environment. When connected the interfaces allow options like stepping and breaking through program code or gathering information on heap usage and thread creation. The profiling option is interesting
for gathering information on the dynamic aspects of the system. We therefore
tried to use the profiling interface together with the iPronto emulation envi-
ronment.

Unfortunately the iPronto virtual machine, Intent JTE from TAO Corporation,
does not provide a profiling interface. To circumvent this problem, we tried to
use the iPronto Java software together with the personal Java reference im-
plementation from SUN. The personal Java reference (pJava) implementation
contains a profiling interface. Using this virtual machine together with the
commercial JProbe[6] tool seemed at first sight a workable solution for profil-
ing the Java software internally.

After setting up the tooling, initial experiments with the test environment show
that starting the iPronto software in the profiling environment fails due to unre-
solved dependencies between the Java software and the Windows platform
layer. The origin of the unresolved dependencies lie in a few additional
iPronto Java packages that use specific platform libraries at run-time. These
libraries are not available with pJava from Sun on the win32 platform and thus
make it impossible to do Java level profiling.

OS level resources
Profiling at the OS level means that the program under test is inspected for
resource usage at the OS level. Operating system resources that can be
inspected are for example processor load, memory usage and network band-
width. Most operating systems support the explicit measurement of these
entities, as a by-product of the fact the OS is a resource manager of the over-
all system or by explicit measurement code that can be installed.

As an example a Linux user can use the `top` command to get an overview of
the some operating system resources, like CPU load and memory usage. Similar the Windows OS provides the `task manager` application to visualize
the CPU load and memory usage. Internally these programs make use of
internal structures of the operating system, which generally are not accessible
to user applications.

In the Windows operating system the system resources are represented by
performance indicators. These performance indicators are stored in the Win-
dows registry. The Windows registry consists of two parts. The first part of the
Windows registry stores the registry keys, persistent and static information
that is used by the OS and applications. This static information can be viewed
via the regedit command. The second part of the registry stores the dynamic performance indicators (stored in memory) that represent system resource usage. The performance indicators are not persistently stored and only the current value of such an indicator can be acquired.

A large set of performance indicators is available in Windows. The information ranges from disk cache and file system to kernel CPU usage and memory management information. A selection of interesting indicators is given in Table 2.

Table 2: Available performance objects of interest.

<table>
<thead>
<tr>
<th>Performance object</th>
<th>Explanation observable data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread(instance)% Processor Time</td>
<td>CPU usage per thread instance.</td>
</tr>
<tr>
<td>Process(instance)\Working Set</td>
<td>Occupied memory of a process (program code &amp; data)</td>
</tr>
<tr>
<td>Process(instance)\Thread Count</td>
<td>Number of threads of the process</td>
</tr>
<tr>
<td>Process(instance)%Processor Time</td>
<td>CPU usage of process instance</td>
</tr>
<tr>
<td>Memory\Available Bytes</td>
<td>Free available system memory</td>
</tr>
<tr>
<td>Processor(0)% Processor Time</td>
<td>CPU usage of the processor</td>
</tr>
<tr>
<td>Objects\Threads</td>
<td>Total number of threads on the system</td>
</tr>
<tr>
<td>Objects\Processes</td>
<td>Total number of processes on the system</td>
</tr>
</tbody>
</table>

3.1.3. Measurement setup

At the beginning of the project no iPronto was available for measurements. To circumvent this issue, measurements are done using the iPronto emulator. The iPronto emulator consists of the Taos Intent Java virtual machine running in the Elate environment, the OS abstraction layer, on a Windows PC.

The measurement machine is a desktop PC with a Pentium4 processor running on a 2 GHz clock frequency with 512 MB of internal memory and a cache size of 512 kB. The desktop PC was cleanly booted and no other applications are running during the test except for the measurement program and the emulator.

The performance indicators selected for observation are given in Table 3. Initial experiments unveiled that the Elate process creates 10 threads. For all Elate threads the processor time is measured. The memory usage and processor time of the encapsulating Elate process is measured. Next, a few system wide indicators are monitored to check if no other activities in the system occur during a measurement.

Table 3: Observed performance indicators.

<table>
<thead>
<tr>
<th>Observed Performance indicators</th>
</tr>
</thead>
<tbody>
<tr>
<td>\PC67240002\Thread(elate/10)% Processor Time</td>
</tr>
<tr>
<td>\PC67240002\Thread(elate/9)% Processor Time</td>
</tr>
<tr>
<td>\PC67240002\Thread(elate/8)% Processor Time</td>
</tr>
<tr>
<td>\PC67240002\Thread(elate/7)% Processor Time</td>
</tr>
<tr>
<td>\PC67240002\Thread(elate/6)% Processor Time</td>
</tr>
<tr>
<td>\PC67240002\Thread(elate/5)% Processor Time</td>
</tr>
<tr>
<td>\PC67240002\Thread(elate/4)% Processor Time</td>
</tr>
<tr>
<td>\PC67240002\Thread(elate/3)% Processor Time</td>
</tr>
</tbody>
</table>
To retrieve the performance indicators a modified version of the pstat tool[7] is used, that allows logging of specific indicators to a file. The results of the measurements in the next paragraphs all show the CPU load and the memory usage of the Elate process. These two values are chosen because they give a good reflection of the platform resource usage.

3.1.4. Scenarios

To profile the memory usage and CPU load of the iPronto in a consistent way, a measurement environment needs to be defined. Next several measurement scenarios are described that each uses the system in a predefined way. The scenarios are setup as a sequence of steps.

Boot scenario

The boot scenario measures the start up of the elate OS abstraction layer, the Java virtual machine and the iPronto software. The remote control application started as default after the boot sequence.

Scenario 1: Remote control switching

This scenario starts where the boot scenario ends. The scenario consists of the next sequential steps that are performed by hand:

- Switch to 4 x RC view.
- Switch to 1 x RC view.
- Switch to 4 x RC view.
- Switch to 1 x RC view.

These steps toggle between the two screen layouts given in Figure 14.
Scenario 2: Remote control and browser switching
Scenario 2 starts from a newly booted iPronto with the remote control application as the active application. This scenario includes the switching between two applications: the remote control application and the web browser application. The steps of the scenario are as follows:
- Switch to web browser application.
- Switch to remote control view.
- Switch to web browser application.

In Figure 15 the screens of the application in this scenario are shown.

Scenario 3: Remote control and multiple browser switching
This scenario also starts with a cleanly booted iPronto. Next it switches between two applications. The difference with scenario two is that the web browser is started with a different start page.

The scenario steps are:
- Switch to browser with google.com as homepage.
- Switch to browser with international.eonline.com.
- Switch to browser with google.com as homepage.
- Switch to remote control application.
- Switch to browser with google.com as homepage.
- Switch to browser with international.eonline.com.
- Switch to remote control application.

The screens of the applications used, as shown in the previous paragraphs, are not shown, as they do not provide more information.

3.1.5. Measurement results

Boot scenario
The results of the boot scenario are given in Figure 16. The figure consists of a CPU load graph and the memory usage. The CPU load figure shows two
graphs, one graph is the processor time of the elate process. The second graph shows the occupancy of the complete processor. In this way, it is possible to distinguish unwanted interference by other processes on the PC.

Figure 16: CPU load and memory usage of the boot scenario.

The CPU load in Figure 16a shows a clear peak on the start of the emulator, followed by a short idle period. Next the CPU is fully loaded with a few short dips. When the emulator is ready, the processor load becomes zero. The memory usage in Figure 16b displays an immediate allocation of 20 MB at start up. Subsequently the memory load increases to approx. 45.5 MB when the emulator is fully loaded and the default remote control application is loaded.

Scenario 1

For scenario 1 the CPU load in Figure 17a give a clear correlation between the scenario action and the CPU load. The CPU load shows a peak at every application switch of the scenario. The width at the base of the peaks is about 1.5 seconds. The memory usage for the first three application switches shows a clear increase. The last application switch does not increase the memory load.
Scenario 2
Scenario 2 uses the web browser as application. When the web browser is started, at about $t=12$, a large block CPU load is started, as can be seen in Figure 18a. The system is heavy busy starting the web browser application. Unfortunately no distinction can be made between start-up of the browser and the processing required for the browser to fetch the default web page, decode the contents, fetch additional data referenced by the html page and render the page.
The next switch to the remote control application shows a small load peak, followed by an idle period when ready. The next peak, a switch to the web browser again, shows two small peaks of 75 and 95% CPU load and low CPU activity of <35% after the peaks. This is considered to be related to the loading and decoding of the web page. On the fourth switch to the remote control application again a small processing peak is shown.

The memory load shows large increase on the first application switch: start up of the web browser. Next small memory allocations are done on the other application switches. Subsequent starting of applications doesn’t show any noticeable increase in memory usage.

Remarkable about the CPU load graph is that after the first start of the web browser application a large amount of CPU activity is observed while the web page is already displayed.

**Scenario 3**

The third scenario is a mix of the previous scenarios and switches between web browser with a different home page. The graphs in Figure 19 do not reveal new insight than what can be expected from the previous paragraphs.
3.2. Streaming analysis

As the intention of the graduation project is to extend the iPronto with streaming video a view on the current iPronto already shows two data streams. The first stream that can be distinguished originates from the LCD display refresh rate. The second dataflow that can be considered as streaming information originates from the wireless network interface. These two data streams are analyzed on some of their characteristics in the next sections. Next conclusions are drawn based upon the analysis.

3.2.1. LCD refresh data

Every time the progressive LCD display is refreshed, the contents of the display are newly rendered. To render the screen information the LCD controller, located on the PXA255, fetches the display contents from a framebuffer in memory. The framebuffer is located in the external SDRAM. The internal organization of the framebuffer depends on the display mode selected for the LCD controller.

Figure 19: CPU load and memory usage of scenario 3.
The LCD controller supports multiple video formats. Firstly it can handle encoded pixel values of 1, 2, 4 or 8 bits that indexes into a palette RAM. The palette RAM is located inside the LCD controller and can store up to 256 entries of 16-bit wide palette information. If the 16 bits per pixel mode is selected, the internal palette logic is bypassed and the pixel data from the framebuffer is directly used to drive the LCD display.

The iPronto uses the 16 bits per pixel mode on a 640 x 480 color display with a refresh rate of 67.5 Hz. The bandwidth used by the LCD controller for the screen refresh is thus:

\[
\text{LCD bandwidth} = \frac{16 \text{ bpp} \cdot 640 \cdot 480 \cdot 67.5 \text{Hz}}{8 \text{ bits}} = 40.5 \text{MB/s}
\]  

(1)

When taking a quick glance at the bandwidth, the system should easily be capable of supporting this bandwidth. The internal 32 bits system bus is running on a clock frequency of 200 MHz and the external 32 bits memory bus has a 100 MHz clock frequency. The external SDRAM is clocked at 100 MHz. In a perfect case these busses support bandwidths up to 800 MB/s and 400 MB/s respectively, which is enough to support the LCD bandwidth. In contradiction to this, the designers of the iPronto have taken additional measures in the wireless driver of the system to circumvent that as a side effect of the wireless traffic, the LCD refresh is corrupted. To explain the interference of the wireless traffic, a more detailed analysis of the LCD refresh traffic is shown hereafter.

The internals of the LCD controller are given in Figure 20. The LCD controller consists of a DMA controller, a set of control registers, palette and dither logic and an in- and output FIFOs. The palette and dither logic is programmable in a 1, 2, 4 or 8 bits per pixel mode. In the 16 bit per pixel mode, deployed in the iPronto, the palette and dither logic is bypassed. The control register contains the memory addresses of the iPronto.
The output FIFO is connected to external electronics that drive the display. The display provides a pixel clock to the LCD controller. The pixel clock is used by the LCD controller to construct a correct timing for driving the LCD panel. The LCD controller constructs vertical and horizontal signals for blanking signals based on programmable counters driven by the LCD clock.

When writing a horizontal line on the panel, the LCD controller writes the pixel data (16 bits) to the display using the pixel clock period $T_{pclk}$. Each $T_{pclk}$ one pixel is written to the display. With $T_{pclk}$ and the capacity of the output FIFO the traffic pattern on the system bus can be calculated. On each $T_{pclk}$ the output FIFO passes one pixel value to the display. The output FIFO obtains its input data from the input FIFO. The input FIFO is organised as 16 entries of 8 bytes wide. As soon as 4 entries of the input FIFO are empty, the DMA controller is signalled to fetch new data to fill up the empty space in the input FIFO.

In the 16 bits per pixel mode, 4 entries of the input FIFO correspond to (4 entries * 8 bytes) / (2 bytes / pixel) = 16 pixels. Thus each $16 \times T_{pclk}$ the DMA controller fetches 32 bytes of data from external memory using the system bus. In the ideal situation, without other traffic on the system bus, this causes the LCD controller to create a repetitive traffic pattern on the system bus. This traffic pattern is depicted in Figure 21 for $T_{pclk} = 40188$ picoseconds and the corresponding refresh rate of 67 Hz of iPronto.
A full input FIFO of 16 entries contains data for $64 \times T_{\text{clk}}$. After $64 \times T_{\text{clk}} = 2.57$ microseconds the FIFO is empty and the LCD controller starts repeating the last pixel until new data is available.

### 3.2.2. Wireless network traffic

To analyze the bus traffic shapes to and from the wireless card, the network stack of the iPronto and the physical PCMCIA interface used by the wireless card is explained. The iPronto uses the Internet Protocol (IP) to connect to external devices and is equipped with a wireless PC card that enables IEEE 802.11b wireless Ethernet connectivity.

The IP protocol in the iPronto uses the wireless Ethernet standard IEEE802.11b for the data-link and physical layer. Several commissions in the IEEE 802 standard body define the Ethernet protocol. Originally intended for wired networking, with several network topologies and addressing schemes, extensions have been made to support wireless networks. This work is carried out under the 802.11 wireless working group of the IEEE and has delivered amongst others the 802.11b standard as used in the iPronto.

The 802 standard distinguishes a data link layer and a physical layer. Common for all 802 standards is the logical link control in the data link layer. The logical link layer provides one uniform interface to the networking layer, in this case the IP layer. Next it interfaces with the Medium Access Control (MAC) layer of the different 802 networks. The MAC layer protocol is different for the diverse physical layers of wired Ethernet versions.

The wireless Ethernet 802.11 standard defines one MAC layer protocol for the different physical layers possible in wireless communication. The physical layer standards 802.11a, 802.11b and 802.11g each apply another set of modulation techniques or use different frequency bands. An overview of the protocol stack is given in Figure 22.

![Figure 21: LCD traffic pattern on the system bus of iPronto.](image-url)
The 802.11 standard defines a common frame format for all physical layers. The frame format consists of a header, a data and a CRC field. The header and CRC field have a fixed length, the data field has a variable length. The length of the data field is specified in the header field.

<table>
<thead>
<tr>
<th>Header</th>
<th>Data</th>
<th>crc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size in bytes</td>
<td>30</td>
<td>0 - 2312</td>
</tr>
</tbody>
</table>

The wireless PCMCIA card applied in the iPronto consists of an 802.11b compliant chipset. The chipset consist of power amplifier, a physical layer chip for (de)modulation and a link layer controller with external memory that autonomously handles the MAC layer protocol. The electrical interface to the host system is designed to conform to the PCMCIA[8] interface standard. The PCMCIA interface connects the link layer controller chip with the XScale PXA255.

The PCMCIA bus interface defines the electrical signals and their timing used for data communication. The electrical signals are logically grouped into three sets. The three sets are the address bus, the data bus and a set of control signals. The data bus supports 8 or 16 bit wide transactions. A read or write operation takes 165 nanoseconds. The control signals consist of read and write signals and a bus grant signal. A Wait signal is defined that indicates, when active, that the slave devices cannot finish the running operation in the current clock cycle. As a result the running bus transaction is extended with one clock cycle. The maximum active duration of the Wait signal is 12 microseconds.

The wireless PC card is directly connected to the external memory bus as a slave device. The XScale CPU initiates all bus transactions to and from the wireless card. The driver from the OS on the CPU initializes the wireless card on start up. The initialization of the wireless card is done by setting values in the control registers of the link layer controller. These control registers are mapped in the memory space of the XScale CPU and accessed by load/store operations from the XScale CPU. Next the link layer controller implements two memory locations that behave like a FIFO queue. One memory location is used for the transmission queue, the other for the reception queue.

On the reception of a complete wireless Ethernet frame the link layer control-
ler signals the host CPU by means of an interrupt signal indicating that the RX FIFO contains data. The interrupt signal is handled by the Linux OS to activate the wireless card driver that fetches the received Ethernet frame and stores it in main memory. For transmission of an Ethernet frame the host driver writes the frame data to the TX FIFO.

3.2.3. Interference LCD and wireless data traffic

The data traffic from the framebuffer in main memory to the LCD controller, for the screen refresh, and the data traffic from the wireless card driver on the XScale CPU interfere when both try to use the shared external memory bus. On the logical level both data streams interfere. As a result the LCD refresh is corrupted and artefacts occur on the LCD display due to the repeated pixels. In the iPronto this issue is solved with a software workaround.

In section 3.2.1 the traffic rate from the LCD controller is explained. When the FIFO buffer of the LCD controller becomes empty, the output driver repeats the last pixel value until new data is available. If the external memory bus is occupied longer than the input FIFO contains data, artefacts appear. The time the input FIFO provides buffering is:

\[
t_{\text{inputFIFO}} = \frac{d \times w \times T_{\text{pclk}}}{\text{bpp}} = \frac{16 \text{ entries} \times 64 \text{ bits} \times 40188 \text{ pS/pixel}}{16 \text{ bits/pixel}} = 2.57 \text{uSec}
\]

Where:
- \(d\) = depth of the FIFO
- \(w\) = width of the FIFO
- \(\text{bpp}\) = bits per pixel
- \(T_{\text{pclk}}\) = pixelclock period

Combining this information from the LCD refresh with the fact that the wireless driver on the XScale CPU performs the fetching of data from the host card with \(T_{\text{PCMCIA}}\) per transaction. This determines the maximum amount of 16 bit PCMCIA transfer before LCD trashing will occur:

\[
\#\text{PCMCIA transfers} = \left\lfloor \frac{t_{\text{inputFIFO}}}{T_{\text{PCMCIA}}} \right\rfloor = \left\lfloor \frac{2.57 \text{uSec}}{165 \text{nS}} \right\rfloor = 15 \text{ transactions}
\]

Where:
- \(T_{\text{PCMCIA}}\) = time per read or write transaction in the PCMCIA address space
- \(t_{\text{inputFIFO}}\) = buffering time of LCD FIFO

The maximum length of successive PCMCIA transaction is 15 operations or 30 bytes. This length is an upper limit, as the time for each transaction can be extended with the WAIT signal on the PCMCIA interface. The PCMCIA specification states that the WAIT signal may be asserted for a maximum of 12 microseconds.

Combining the maximum length of a wireless Ethernet packet of 2346 bytes with the above information shows that interference will result if no additional measures are taken. In the iPronto the software driver of the wireless card is
adapted to deal with this situation. It splits up each transaction for the wireless card FIFOs into multiple transactions with intermediate wait states. The intermediate wait states prevent interruption of the data stream to the LCD controller.

Next to the current software workaround other solutions for this problem can be the usage of a different electrical interface standard or changing the applied arbitration in the memory controller. The first option would result considerable changes in the hardware design and can be an option for a newer version. The second option is not possible as the internal arbitration scheme of the system bus in the PXA255 is neither programmable nor specified by Intel.

3.3. Conclusions

In this chapter we show how the various Java threads map into a process on the operating system level. Measurements on the iPronto software stack (using a desktop PC) show CPU loads of a few seconds on application switching. The processing load of the current applications is event driven on response to user input. The web browser application shows a large CPU load on start-up, even after the web page is displayed.

An analysis of the current data streams in the system, the LCD screen refresh and the data from the wireless network card, show periodic data transfer with a small grain size and large throughput. The current buffering in the LCD controller is small and artefact on screen occur when the external memory is occupied too long by other system components. Additional measures have been taken by the iPronto designers to circumvent the interference.
4. Conceptual architecture

This chapter will firstly define the top-level architecture. This top-level architecture will gradually be refined to the software and hardware of the different subsystems. The concepts described in this chapter are broader than the requirements of section 1.2. They can in general be applied to systems that should support streaming media.

4.1. Separation streaming and control domain

The current iPronto application can be classified as 'control' oriented, as the applications have a static nature and are event-driven on response to user input. Next to the event driven processing, a few data streams run at the platform under control of the OS. As shown in chapter 3 these two streams interfere at the memory that acts as a shared resource.

To extend the iPronto for applications with streaming media, like playing MPEG-2 or MPEG-4 movies via a wireless interface, we propose to separate the system into two domains. The first domain handles all event driven processing and is called the control domain. The second domain is the streaming domain and handles all stream based processing. Both domains have a defined interaction though are strongly decoupled.

The current iPronto architecture is a mix of control and streaming. The largest part of the iPronto architecture fits in the control domain and is extended with a separate subsystem that handles all processing of streaming data. The separation into two distinct domains enables exploiting the different properties of both domains and better reasoning about the system performance.

The processing functions of the streaming environment are expressed in a task graph. The complete task graph acts as a processing pipe. The processing pipe consists of a set of tasks that each consume and produce a flow of data. The tasks have dependencies. Each task has its own private memory. Tasks do not share memory. Multiple references to data are in general not necessary. The tasks in the graph represent the computation functions that
run simultaneously. The edges represent in-order communication between the tasks. Each task executes when tokens are available on all its inputs.

![Task graph representing the processing pipe.](image)

Figure 25: Task graph representing the processing pipe.

The control and streaming domain communicate by means of mode settings and status information. Mode settings can be further classified into parameters for the tasks and configuration information of the topology the signal-processing graph. Parameters are control oriented. An example of a parameter is the value for a brightness function. The configuration information is the topology of the signal-processing graph. The signal-processing graph consists of a set of tasks mapped on computation resources and the description of the communication path between the computation resources.

### 4.2. Hardware architecture

The communication between the two domains is implemented via shared memory. For a unified memory architecture the basic idea behind the separation in two domains is the fact that both domains have different requirements on the memory accesses. The event driven control domain is characterized by irregular memory transfers with a short latency as primary requirement. In the streaming part of the system the memory transfers are characterized by periodic requests that need a large throughput and have less constraints on the latency.

Figure 26 shows the top-level hardware architecture of the two domains together with the background memory. The shaded part of the memory system represents the memory arbiter. The arbiter controls the memory access requests of the control and streaming domain. The memory arbiter uses an algorithm to satisfy the different memory access requirements of both domains. The algorithm for the arbiter is described in [9]. It minimizes the latency for the control domain and guarantees the throughput for the streaming domain. As latency is not a hard constraint for the streaming domain, the throughput can be achieved with sufficient buffering.
For portable devices, like the iPronto, the available battery energy is limited. To achieve an acceptable battery time, one should strive for a very low energy usage by the system. Low energy usage is in contradiction with the current clock frequency race of modern PC systems. To achieve a low energy consumption the clock frequency should be kept as low as possible since the power consumption of a digital CMOS circuit is a linear function of the clock frequency. Another option to reduce the power consumption is to lower the voltage of the circuit. A low supply voltage also implies lower power consumption, power consumption is quadratically related to the supply voltage. This is one of the motivations for the approach of computing in space, discussed hereafter.

For the hardware model of the streaming domain we consider a set of processing units interconnected by a communication network. The communication network represents a bus, switching circuit or any other type of communication circuit as long as it can give guarantees on end to end connections between processing units. A processing unit (PU) is a combination of computation hardware, local memory and an interface to the communication network. The communication path set-up between two processing units gives the throughput similar to a circuit switched connection. For the case of a bus, guarantees on throughput can be given using a time multiplexing scheme\(^2\).

Figure 27 gives the hardware template of the streaming domain. The number of PUs is flexible and can be tailored to a specific platform instance. A connection to the shared memory provides access to a large background memory. The PUs contain local memory for the storage of instructions, temporary data of its task(s) and buffering for communication. The local memory should maximize the locality of reference principle. The assumption is that a PU contains all instruction code necessary to perform its task(s). Once a task is configured on a PU all instructions are fetched from the local memory. During configuration of a task, the instruction can be fetched from the shared memory. All temporary data necessary for the processing is stored in the PU memory. After configuration the communication network is only used for pass-

---

\(^2\) Example: the IEEE1394 bus uses a time-multiplex scheme to give guarantees for isochronous data-traffic. The bookkeeping of resource allocation for isochronous data-traffic is one centrally per bus. The (cyclic) bus arbitration scheme guarantees priority for isochronous traffic before asynchronous traffic.
ing data between tasks on different PUs.

\[ PU \rightarrow PU \rightarrow PU \rightarrow Mem \text{ interf.} \rightarrow \text{Connection to shared memory} \]

\( PU = \text{Processing Unit} \)

*Figure 27: Template of the streaming domain hardware.*

The computation hardware of in the PUs can be standard RISC cores, application specific VLIW processors, reconfigurable hardware (FPGAs) or dedicated hardware. The heterogeneity provides a trade-off between flexibility, cost, time to market and it allows tailoring to a specific domain.

4.3. **Software architecture**

This section will firstly discuss the control domain abstraction of the streaming domain. The goal is to create a configuration image for the streaming domain. Furthermore the relation to the hardware architecture is presented.

The nature of the streaming applications, like MPEG-2 decoding or MPEG-4 video decoding, is a flow of data through a processing pipe. The software architecture used for representing the processing pipe should reflect this main characteristic. This streaming abstraction is used to construct the streaming application.

The software engineers that construct the flow graph should not be concerned with signal processing. Their effort should concentrate on the construction of the user interface concepts and the inter-device connectivity options of future products. It is preferable that the processing graph is constructed from predefined elements. The composition of these elements should deliver a predictable performance of the processing graph.

A *service* is a software entity that provides a meaningful and coherent activity, a building block to the application. It specifies the behaviour in high-level terms. Each service groups a set of streaming components that together form the design entity at a higher level of abstraction. This simplifies the design of the other functionality in the middleware as the specifics of the platform instance are hidden. A service 'MPEG-2 decoder' could be composed of several components like a demultiplexer, a video decoder and audio decoder. A service is an intermediate layer in the software stack.

A *component* is an abstraction in the control domain of a media processing function in the streaming domain. The realization of the processing function of a component is not visible to the service and can be implemented in software.
or hardware. The components have a reference to one or more tasks that implement the processing functionality.

![Diagram of services and components in the SW stack.](attachment:diagram.png)

*Figure 28: Streaming services and components positioned in the SW stack.*

The components are offered by the platform instance. If the platform instance changes, the interfaces provided by the components do not change and thus the interface between the service and the components stay the same. The placement of the services and the components in a software stack is illustrated in Figure 28.

To illustrate the services an example is given in Figure 29. The graph in Figure 29a shows the basic dataflow pipe at the service level, though informal. The number of inputs and outputs of a resource, the data format(s) supported by each input and output and the parameters to control the processing options of the component should be made explicit.

A more formal approach is given in Figure 29b. Each service, drawn as a rectangular box, has a defined number of inputs and outputs. Each input or output is represent by a pin with properties. The properties of the pin are the direction, data format and the maximum data rate. The parameter interface depict the 'control' parameters as described in section 4.1. Graphically this is indicated with the UML notation of a circle connected to the component annotated with the name of the corresponding interface.

![Example graph](attachment:example_graph.png)

(a) Requirements graph
Channels are connected to one output pin and one input pin. Channels implicitly imply a method to move data from one input to one output. They model the data transport protocol. The data transport protocol is hidden for the software that constructs the graph at the service level. The specific data transport protocol is addressed at the component level.

The service layer models the resources for streaming media on a high level of abstraction. It builds upon the components provided by the platform vendor. The rest of this section will deal with the component interaction towards the streaming domain.

As explained, components model the streaming pipe in the control domain. Each component has one or more tasks representing the functionality in the streaming domain. The tasks have one or more references to the implementation of the processing function in the streaming domain. Each task is at least mapped to one computation resource in the streaming domain. The computation resources are the processing units of the hardware template in Figure 27.

The manufacturer of the platform delivers a library of components in combination with a driver to address the streaming hardware. The driver is specific to the OS running on the hardware of the control domain. Such a library could be provided by the manufacturer of the streaming domain hardware, or by a third party software provider.

The designers of a component have several options to construct a component. Firstly they can design the component from scratch, including the task(s) in the streaming domain. Alternatively they can use a set of predefined components with the accompanying tasks from a library to construct the new component. The possibility to use a set of predefined available components promotes re-use across designs. Missing combinations of components/leaves can be newly designed to complement the required functionality.

The processing function of a component is decomposable into one single or multiple tasks. This can be achieved in several ways. A component could directly represent several processing tasks or it uses hierarchy to hide complexity. Several options of components and tasks are given in Figure 30.
If a component contains hierarchy, like component C in Figure 30, the component encapsulates control software that delegates control parameters to the correct sub-component. Component D in Figure 30 shows the possibility that a component can have two task sets associated, both representing the same functionality, though with different resource usage of the processing units.

![Control domain](image1)

![Streaming domain](image2)

*Figure 30: Several possibilities of component / task combinations.*

In current practices the designer of a component will design one optimal solution of the 'processing function' for the available platform hardware. The processing function is partitioned over the available hardware in such a way that the partitioning maximizes the performance. For systems on chip (SoC) with multiple programmable units in a system that have supports multiple concurrent running applications, component designers need to provide alternative mappings. The alternative mappings provide a different partitioning of the tasks over the platform hardware. The partitioning is done at run-time, dependent on the applications running on the platform.

To enable a run-time decision each task is annotated with usage figures for the processing units on which the task can be executed. As an example Table 4 gives a description of the processing load from each task in Figure 30. Several tasks can run on multiple resources, and some tasks are only available on specific units. Besides the processing load of a task other parameters like the occupied instruction and data memory and the data traffic of each output should be taken into account.
Applications may want to add or change the processing graphs while multiple graphs are running on the platform. Changes in the graph configuration should be seamless to the user. A simple method for constructing the graphs and finding a mapping that runs on the hardware should be available.

Summarized a method is needed to:

1. (re)configure the graph.
2. map the new graph on the available processing units.

The solution should take into account:

- Layered software approach to apply strong decoupling in the software stack.
- Resource usage of running applications on the processing units.
- Specification of the communication network.

To address these issues, a graph manager and a resource manager are introduced at the component level in the SW stack. Both do not represent a task in the streaming domain, they serve as common facilities for the components. Figure 31 show the position of these two facilities in the SW stack.

To support the decoupling in the software stack the notion of a graph manager is introduced. The graph manager is the access point to retrieve information on the available components in the system. The graph manager provides a unified way for the components to communicate with the corresponding tasks in the streaming domain. The graph manager centralizes the communication with the streaming domain and hides the communication mechanism from the components. The communication contains the task graph topology and the task control parameters.
Figure 31: Placement graph and resource manager in the SW stack.

The components can be connected via the graph manager. The graph manager assists in connecting components. It performs a format check on the pin type of components to connect in order to validate the correctness of the requested graph. It also keeps track of the flow graphs running in the platform. This is necessary as the scope of the application is limited, applications do not have information on the running graphs of other applications. As soon as a new flow graph is constructed in the control domain, the graph manager interacts with the resource manager to retrieve a new configuration for the streaming domain.

The resource manager deals with the distribution (mapping) of the processing graph over the processing units in the streaming domain. The new configuration is based on the current tasks and the new tasks running in the streaming domain. The resource manager keeps track of all resource usage in the streaming domain. It uses a description of the streaming domain that enumerates all processing units and the interconnect structure.

Tasks have input and output ports that have identical properties (like the data format) with the input and output pins of the component in the control domain. The communication model between the ports of two tasks is based on a FIFO queue with defined operations for putting and fetching data token from the queue. All tasks use the same semantics in the operations on the queue. If different semantics are used by various task implementations, the pins of the components in the control domain could be labeled with a field indicating a type. The graph manager could be used to decide on the values of the labels to introduce a 'conversion' function between the ports of two tasks with a different semantics. Further investigation should be done to see if this is a viable method. It is advisable to use only one set of semantics.

To start a component it is not necessary that all input and output pins of a component are connected. Some pins may be dangling. To support this functionality, a task should support state. Each state represents different combina-
tions of active input and output connections. The component can control the state of a task, but the state of the task is not exposed to the application using the component. As an example, illustrated with Figure 32, we will consider an example where the user starts the playback of a video from a network source.

The example in Figure 32 consists of two parts. The first part, in Figure 32a, depicts the initial state of the device when it is switched on. A graphics renderer is used to draw the user interface. The graphics renderer is connected to the display that represents the screen. The audio feedback player is used to generate user feedback sounds. The display and speakers both contain multiple input pins to support mixing of multiple sources.

Now assume the user starts the playback of an MPEG-2 encoded video from a DVD or a server in the network. For the reception of the encoded DVD data a network interface receiver is connected to a MPEG-2 decoder. The network interface represents the reception of the streaming media from the network and handles the control protocol between server and the client device. The network interface is connected to a MPEG-2 decoder that parses and decodes the streaming video. It uses an MPEG-2 elementary stream as input and provides video and audio input using standard video and audio formats. The MPEG-2 decoder outputs are connected to the speakers and display.
components respectively. The configuration of the components when playing the DVD is given in Figure 32b. It shows that the unconnected pins of the display and speakers in Figure 32a are connected to the MPEG-2 decoder.

As the number of pins of a component is static and is determined at design time, the pins give a bound on the complexity of the applications that can be constructed. The platform in Figure 32 supports three video streams and two audio streams simultaneously.

4.4. Conclusions

In this chapter we introduced the separation of the system in a streaming domain and a control domain. Both domains interact by means of configuration and status information. Separation into two distinct domains enables the exploitation of the different memory characteristics of the memory accesses. The streaming domain is characterised by memory transfer with a large throughput, the memory transfers of the control domain are characterised by short latencies. Both systems can interact on one common memory system through an arbiter that satisfies these requirements.

The template of the hardware domain consists of a set of processing units. A processing unit is tailored for a specific function but still programmable. Once configured the processing units work autonomously on local data. No background storage in memory is necessary for computation or instructions. The processing units have an interconnect circuit that can give guarantees on the throughput between two end points. The guarantees for throughput of an end to end connection are configured at run-time. This multi-processor approach should enable very low power consumption. Further research should determine if applications can be modelled in such a way that they fit on this architecture.

The control domain uses a service layer to model the streaming pipe in the streaming domain. The service layer represents coarse grain functions like MPEG decoder(s), speakers and display as abstraction. The services are composed of one or more smaller components (objects) representing the functionality of the service. Every component is linked with one or more tasks implementing the processing functionality in the streaming domain. The set of tasks provided can differ in processing and memory load or target a different type of processing units in the streaming domain. This provides the possibility to partition the task graph over the processing units in the streaming domain.

To control the connection of flow graphs on the component level a graph manager is introduced in the software stack of the control domain. The graph manager has knowledge of all running application graphs and provides a single point of interaction to set-up graphs at the component level. The graph manager uses a resource manager to select the best configuration for the streaming domain. The resource manager uses a description of the streaming domain that consists of a description of the capabilities of each processing unit and a description of the interconnect structure. The component graph together with the leaves annotated with the resource usage enables the re-
source manager to determine the best configuration under the constraint of the available processing units and the corresponding leaves.

**Related work**

The modeling of a flow graph into objects or components with pins and a transport protocol is a well-known principle. Existing frameworks that use this abstraction are DirectX, GStreamer and TSSA. DirectX and GStreamer are both PC based solutions for Windows and Linux respectively. TSSA is an embedded streaming framework for the TriMedia processor. Common to all these three solutions is that they apply a mapping of component(s) on an operating system thread that implements the processing functionality in software. The operating system thread is created by the framework and the scheduling of the tasks is done run-time using a priority driven scheme. Support for hardware acceleration is available by means of threads that use shared memory for data transfer, memory mapped I/O and interrupts to communicate with the hardware. When a component is accelerated by dedicated hardware the thread implementing the functionality is in a blocked state while the hardware is processing. In the meanwhile the scheduler can schedule other threads that are ready to run.

The DVP-API2[10] is a control interface for a streaming subsystem, abstracting from different possible streaming architectures and implementations. It provides similar functions like components and the connection of components. It completely abstract from the fact if a processing task is performed in software or hardware. A platform using the DVP-API2 provides a fixed set of graphs at the component level. The software on top of the DVP-API2 selects a specific use-case of these graphs, after which the platform is reconfigured. All use cases are currently constructed and tested at design time of the platform, no run-time resource management is done.
5. Consequences for iPronto

The current iPronto design has no support for streaming video. The extensions necessary to enable streaming media are described in this chapter. The logical device graph, introduced in chapter 4, is positioned in the software stack. Next an extension to simplify the programming model for applications is discussed in section 5.1. Options for the hardware extensions are discussed in section 5.2.

5.1. Service layer

The streaming service level, introduced in section 4.3, should be positioned in the software stack of iPronto. The software stack of iPronto, see Figure 4, is set-up in packages that group software with common functionality. To introduce the services a new package 'streaming services' is proposed that groups all services that support streaming. This package will also include the graph manager as it is closely coupled to the streaming services. The streaming services should be accessible directly by an application, or a by other services that use the streaming resources to construct intelligent functionality for applications.

![Figure 33: Placement streaming services in the iPronto software stack.](image)

From an application point of view multiple methods exits to construct streaming media graphs. An application could construct a flow graph itself or it could delegate the construction of the flow-graph to another (middleware) service. Using a middleware service to construct the streaming graph decreases the complexity of the application and enables reuse over multiple applications. Furthermore the introduction of a single point of interaction for all streaming media eases the management of the streaming domain.

The iPronto applications and middleware services are all programmed in the Java language. The streaming services providing access to the multimedia capabilities of the underlying platform should therefore be available in the
Java language. Additionally the usage of a standardized and industry wide accepted API is preferred over a proprietary solution.

The widely known API for streaming services in Java is the Java Media Framework (JMF) specified by Sun Microsystems [11]. The JMF targets the desktop market and implies the use of the J2SE specification. As shown in Figure 8 this means a large footprint. The industry became aware of this and created the Mobile Media API[12] through the Java Community Process[13]. The Mobile Media API (JSR-135) targets the J2ME, the Java VM used in the iPronto. The Mobile Media API (MMAPI) has a smaller footprint and defines less APIs then JMF, though uses the same basic properties. One of the results of the smaller supported API set is the smaller footprint.

Description of the Mobile Media API
The MMAPI package separates multimedia processing into two parts:

1. The data delivery protocol.
2. The data content handling.

The data delivery control handles the fetching of the data from its location (hard disk, web server, etc) into the system. The data content handling defines the parsing, processing and rendering the data to output devices like speakers and displays. The APIs specified by MMAPI are agnostic of a specific media type. The API can handle different multimedia types like MP3 music, MPEG-2 or MPEG-4 video with the same function calls.

The data delivery protocol is represented by the `DataSource` object, the data content handling is represented by a `Player` object. Both objects encapsulate the two parts of multimedia processing defined by MMAPI. To create a Player object, a factory pattern[14] named `Manager` is used. The manager takes care of creating a Player object for a specific media type. The Player object is not defined as a concrete class though with an interface. The manager instantiates an object that implements the player interface for the specified media type. The manager is an intermediate that provides access to an implementation specific mechanism for creating players.

![Diagram of the main MMAPI classes and interfaces.](image)

The main interface and classes of the MMAPI are given in Figure 34. Both the
Player and DataSource object implement the Controllable interface specified by the MMAPI. Via the generic Controllable interface the application(s) can query the available Control methods of the specific DataSource or Player object. Each specific DataSource or Player object is allowed to implement different methods for controlling the media playback.

The MMAPI defines three methods for an application to create a player object. The methods differ in the description of the content source. The three methods are:

1. Creation from a media locator.
2. Creation from a DataSource.
3. Creation from an InputStream.

1) The first method takes a media locator as input. A media locator is a string that contains the description of the media in the form of a Uniform Resource Identifier (URI)[15]. A Uniform Resource Identifier (URI) is defined as a compact string of characters for identifying an abstract or physical resource. A URI is defined in the form:

   <scheme>:<scheme-specific-part>

The <scheme> part of the URI specifies the 'protocol' to be used to retrieve the data. Table 5 lists the schemes defined by MMAPI. The <scheme-specific-part> specifies the protocol specific parts and is defined separately.

   Table 5: Schemes defined by MMAPI.

<table>
<thead>
<tr>
<th>Scheme type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>Data fetch with the Hyper Text Transfer Protocol.</td>
</tr>
<tr>
<td>RTP</td>
<td>Data fetch using the Real Time Protocol.</td>
</tr>
<tr>
<td>capture</td>
<td>Captures data from a local recording device. (eg. an camera or microphone).</td>
</tr>
<tr>
<td>file</td>
<td>Fetches data from the local file system.</td>
</tr>
</tbody>
</table>

A URI is a superset of the well-known Uniform Resource Locator (URL). A URL identifies resources via a representation of their primary access mechanism, their network 'location'. In the context of the MMAPI, the resource represented by the URL is the media locator. As example a URL could be "rtp://dvdplayerliving.myhome.com/moviename.mpeg2".

2) The usage of a DataSource object for the creation of a player enables the application to implement a custom object to handle an application-defined protocol. This is a very flexible solution, but it burdens the application developer with the implementation of the specific protocol.

3) The third version of the createPlayer method creates a Player from an InputStream. This can be used to interface with other Java API's that use InputStreams. An InputStream is abstract class that serves as parent for a range of classes that represent a stream of bytes. It should be noted that InputStream does not provide random seeking functionality. So a Player cre-
ated from an InputStream can only playback content from the beginning of the content.

**Life-cycle management**

To provide life-cycle management the Player object has five states. The purpose of these life-cycle states is to provide control over potentially time-consuming operations. The states of the Player object are described in Table 6. The state machine of the Player object is given in Figure 35.

**Table 6: State description of the Player object.**

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unrealized</td>
<td>No content type information available and no resources allocated.</td>
</tr>
<tr>
<td>Realized</td>
<td>Information on content type available, no resources allocated.</td>
</tr>
<tr>
<td>Prefetched</td>
<td>All required resources are allocated.</td>
</tr>
<tr>
<td>Started</td>
<td>Running, processing content.</td>
</tr>
<tr>
<td>Closed</td>
<td>Player object released all resources in use.</td>
</tr>
</tbody>
</table>

To use a Player parameters must be set up to manage its movement through the states and then move it through the states using the Player's state transition methods. When a Player is first constructed, it's in the UNREALIZED state. Transitioned from UNREALIZED to REALIZED, the Player performs the communication necessary to locate all of the resources it needs to function (such as communicating with a server or a file system). The realize method allows an application to initiate this potentially time-consuming process at an appropriate time.

![Figure 35: State machine of the player object.](image)

Typically, a Player moves from the UNREALIZED state to the REALIZED state, then to the PREFETCHED state, and finally on to the STARTED state. A Player stops when it reaches the end of media; when its stop time is reached; or when the stop method is invoked. When that happens, the Player moves from the STARTED state back to the PREFETCHED state. The Player is then ready to repeat the cycle.

**Usage of the MMAPI for iPronto**

For the iPronto the implementations of the streaming functionality should be (very) low power. As explained in chapter 4.2 we would like to achieve this by a computing in space approach. Therefore the processing of the streaming media should not be done in middleware layer of the system. The computationally intensive part should be done in the streaming subsystem of the plat
form. To enable effective use of the streaming subsystem the Player and DataSource objects should be used as an intermediate step for creating a graph at the service level as in Figure 29.

Application developers should only use DataSource objects provided by the specific MMAPi package on iPronto. If applications use self-developed DataSources it would hamper the processing efficiency. This is explained as follows: The streaming data is ‘received’ by the DataSource at application level in the Java part of the system. The processing of the data is located in the streaming subsystem, the data needs to be transferred to the streaming subsystem. The transfer implies the traversal of the border from the Java virtual machine to the ‘native’ OS layer. The method for this is the Java native interface. Data transfer to the streaming subsystem by using a Java to native interface is expensive with the current version of the Java virtual machine on the iPronto. It is therefore advisable to limit the use of DataSources and restrict the usage to a set of predefined DataSources that proxy the reception of data on the native level.

Using the requirements outlined in section 1.2 a set of Player objects can be identified. The proposed Player objects are given in Table 7. For each supported format a dedicated player is constructed.

<table>
<thead>
<tr>
<th>Class name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPEG2Player</td>
<td>Playback of audio and video in the MPEG-2 format.</td>
</tr>
<tr>
<td>MPEG4Player</td>
<td>Playback of audio and video in the MPEG-4 format.</td>
</tr>
</tbody>
</table>

Each Player class embeds the information on the specific graph necessary at the service level. Next it delegates the set of control interfaces it provides, via the Controllable interface, to the specific service that implements the functionality.

**Streaming services**

Section 4.3 explained that the streaming services are software abstractions of the processing functionality. The streaming services have a large grain size. This enables the middleware software to construct simple graphs of only a few services. Typically only a source, a media decoder and a sink are modeled.

The services necessary for playback of video are given in Table 8. The table contains sink services like speakers and the display, network interfaces acting as sources and media decoders for several specific media formats.

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display</td>
<td>Models the screen and graphics mixer</td>
</tr>
<tr>
<td>Speakers</td>
<td>Models the build-in speakers</td>
</tr>
<tr>
<td>MPEG-2 decoder</td>
<td>Decoder for MPEG-2 video</td>
</tr>
<tr>
<td>MPEG-4 decoder</td>
<td>Decoder for MPEG-4 video</td>
</tr>
<tr>
<td>HTTP network interface</td>
<td>Reception of data over a HTTP connection</td>
</tr>
<tr>
<td>RTP network interface</td>
<td>Reception of data via the RTP protocol</td>
</tr>
</tbody>
</table>
Interaction MMAPI, streaming services and streaming subsystem
The state transitions methods of the player object can be coupled to actions on the graph at the service level, the components represented by the services and the effects on the streaming subsystem. In Table 9 the effect of the different subsystem are coupled.

At the transition from the Unrealized to the Realized state the graph service level is created. This means that for each content type, i.e. specific data format, a corresponding logical device graph is known. The designer of the streaming abstraction can do this in two ways. Firstly for all supported data formats a corresponding graph is made at design time. This results in a (large) set of media formats and the corresponding logical graph. A simple selection method can determine the required graph. A disadvantage is the fact that only streaming components known at design time of the MMAPI package can be supported. The second option involves the usage the `Graph Mapper` concept[16]. The Graph Mapper automates the selection and creation of the logical graph at run-time. Besides the in-box streaming the Graph Mapper can deal with the streaming pipe in the network outside the device. For the support of only a few logical graphs the introduction of a Graph Mapper might be too heavyweight. A simple selection statement between several graphs suffices in that case.
Table 9: Player state transaction coupled to the services, components and the streaming domain hardware.

<table>
<thead>
<tr>
<th>State Transition</th>
<th>Actions performed on services graph</th>
<th>Actions at component level</th>
<th>Streaming domain hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>realize</td>
<td>Services are connected.</td>
<td>Creation of logical graph via graph manager.</td>
<td>None</td>
</tr>
<tr>
<td>prefetch</td>
<td></td>
<td>Mapping of logical graph onto streaming subsystem via the resource manager.</td>
<td>Reservation of PUs in streaming subsystem. PUs acquire instruction code. Setup of point to point connections between PUs. Start of intermediate tasks.</td>
</tr>
<tr>
<td>deallocate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>start</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>close</td>
<td>Disconnection of service graph.</td>
<td>Release of all acquired resources.</td>
<td>Tear down of reserved communication paths. Release of acquired processing units.</td>
</tr>
<tr>
<td>MMAP interface</td>
<td>Streaming services package</td>
<td>Streaming subsystem package</td>
<td>Hardware streaming domain</td>
</tr>
</tbody>
</table>

Placement in Figure 36.

Empty fields indicate that the actions are delegated to lower layers in the software stack, in this table towards the next right-hand side column.
Overview of the proposed software extensions

To summarize, Figure 36 gives an overview of the iPronto software stack with the proposed extension to support media decoding.

At the service level a MMAPI package is introduced. This package supports the MMAPI specification and depends on the Streaming services package. The streaming service package contains all the proposed services that model the streaming behavior in the system, as identified in Table 8. Using a separate package that contains the streaming services makes it more easy to for future services to use the streaming services.

Next, between the Java layer and the operating system layer, a package ‘streaming subsystem’ is proposed. This package contains the components, the abstractions of the media processing in the control domain, as described in section 4.3.

The components in the streaming subsystem use a driver of the operating system layer to communicate with the actual hardware subsystem. The driver is specific for the specific hardware subsystem.

5.2. Options for hardware extension

This paragraph discusses several hardware options to expand the iPronto with a separate streaming subsystem. The development of an integrated circuit is not viable for the iPronto development group. The scope of this section is thus limited to off the shelf, commercial available, products.

The hardware extension can be classified into two distinct parts: changing the wireless Ethernet hardware to support higher bit-rates and next adding video processing functionality.
5.2.1. Wireless network interface

The current wireless Ethernet card in the iPronto, an Intersil Prism2 chipset, supports the IEEE802.11b standard. The IEEE802.11b standard supports channel data rates up to 11 Mbit per second. Higher up in the network stack, at the internet protocol, the usable data rate is limited to approximately 6 Mbit per second. This data rate is not sufficient for supporting playback of DVD material. As stated in Table 1, the MPEG-2 material on a DVD has a maximum data rate of 10 Mbit per second. Therefore it is necessary to switch to a different wireless Ethernet standard.

Possible options for a different Ethernet standard are the IEEE802.11a or IEEE802.11g standard. The IEEE802.11a standard supports up to 54 Mbit per second in the 5 GHz band, the IEEE802.11g standard supports 20+ Mbit per second in the 2.4 GHz band.

A very interesting dual chip solution is the Philips SA5250 IC together with the SA5251 IC[17]. This chipset support all three wireless Ethernet standard, IEEE802.11a/b/g, and has a programmable PHY/MAC layer. This enables to support upcoming encryption standards in the wireless domain. In addition to the standard MiniPCI, PCI, and Cardbus interfaces, the chipset provides a general-purpose 32-bit interface for direct connection to an embedded host processor. This chipset is supported by Linux drivers and can thus easily replace the current wireless card.

5.2.2. Video processing functionality

Firstly we will discuss the required processing power of an RISC core for VGA sized MPEG-2 and MPEG-4 decoding. Next the usage of field programmable gate arrays (FPGAs) with embedded RISC cores is explored.

**RISC core**

ARM ltd. provides a software MPEG-4 codec for its ARM9 family of processors. The CPU load figures, according to ARM[18], for decoding of MPEG-4 video, with a QCIF resolution (176 x 144 pixels) and 15 frames/second are given in Table 10. The figures of the VGA format (640 x 480 pixels; both 30 and 60 fps) are calculated using the QCIF load figures from ARM.

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU load [MHz]</th>
<th>QCIF format 15 fps</th>
<th>VGA format 30 fps</th>
<th>VGA format 60 fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>10</td>
<td>242</td>
<td>485</td>
<td></td>
</tr>
<tr>
<td>Error concealment</td>
<td>2</td>
<td>48</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>De-block</td>
<td>20</td>
<td>485</td>
<td>970</td>
<td></td>
</tr>
<tr>
<td>De-ring</td>
<td>6</td>
<td>145</td>
<td>291</td>
<td></td>
</tr>
<tr>
<td>Image scaling</td>
<td>12</td>
<td>291</td>
<td>582</td>
<td></td>
</tr>
<tr>
<td>Colour conversion</td>
<td>9</td>
<td>218</td>
<td>436</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>59</td>
<td>1430</td>
<td>2861</td>
<td></td>
</tr>
</tbody>
</table>

*Table 10: CPU load MPEG-4 software decoder for an ARM9 processor*
Although not all functions in Table 10 are directly related to MPEG-4 decoding, some are on image enhancement, others on conversion and images scaling, they give a good idea on the necessary processing requirements. The VGA sized numbers are probably optimistic, as the performance of the memory hierarchy, an ARM9 uses caches, doesn't scale as easily as the CPU frequency. This leads to the conclusion that the usage of a ARM9 core does not satisfy the application requirements.

Field Programmable Gate Arrays

Current trends in FPGAs, like embedded RISC processors, increasing number of logic elements, on board RAM and integrated memory controllers make it a very interesting candidate. The flexibility with FPGAs to change a design in a late stadium, necessary to support late specification changes, is very interesting.

The possibility to use embedded RISC cores, like an ARM9 core in the Altera Excalibur FPGAs or up to four PowerPC cores in the Xilinx Virtex-II FPGAs, enables a very interesting option for hardware/software co-design. FPGAs with embedded RISC cores enable starting the design with a description of the functionality in the C language. This C level entry for the design can gradually be refined in combination with the design of the programmable hardware part of the FPGA. The compute intensive parts of the design can be treated by special purpose hardware, mapped into the programmable hardware of the FPGA. This approach is fast, flexible and thus very interesting.

One of the main drawbacks is the cost of an FPGA with an embedded RISC core. As an example Table 11 gives a summary of two members of the Virtex-II Pro FPGA[19] family from Xilinx Corporation. It summarizes the number of PowerPC cores available and several numbers on the size of the programmable hardware in the FPGA. Two device types are mentioned in the table. Both devices types are available in multiple versions. The versions differ in the IC package and the corresponding number of IC pins available for user I/O.

<table>
<thead>
<tr>
<th>Device type</th>
<th>PowerPC Processor Blocks</th>
<th>Clock freq. PowerPC (MHz)</th>
<th>Logic cells</th>
<th>Slices</th>
<th>18 x 18 bit multiplier blocks</th>
<th>Max. Block RAM (kbit)</th>
<th>Production price 2H2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC2VP7</td>
<td>1</td>
<td>300/350/400</td>
<td>11088</td>
<td>4928</td>
<td>44</td>
<td>792</td>
<td>$59.70</td>
</tr>
<tr>
<td>XC2VP20</td>
<td>2</td>
<td>300/350/400</td>
<td>20880</td>
<td>9280</td>
<td>88</td>
<td>1584</td>
<td>$124.00</td>
</tr>
</tbody>
</table>

The production price column of Table 11 reflects the prices of the most inexpensive FPGA versions with a 300MHz Power PC. These versions relate to the package with the smallest number of user I/O pins. To illustrate the capacity of a Virtex FPGA, the firm Amphion reports on a main level at main profile MPEG-2 decoder[20] using 7377 Slices and 72 kilobit of block RAM from a Virtex FPGA.
Media accelerators
Several vendors provide solutions for mobile multimedia decoding. Many solutions target the UMTS mobile phone market. Common to all the solutions for the mobile phone market are small video formats, like CIF and QCIF, and low frame rates (up to 15 frames per second).

An investigation to chips that support MPEG-4 Advanced Simple Profile and/or MPEG-2 Main Level at Main Profile led to ICs. Table 12 summarizes the main characteristics of the IC's. Next to the capabilities of MPEG-2 and MPEG-4 decoding the table summarizes:

- Possibility to directly connect a LCD screen.
- Size of the external SDRAM memory.
- Bus type to connect to a host processor.
- Power consumption.

It is very hard to perform a benchmark on the performance of the different chipsets. Manufacturers often do not describe in detail the exact conditions under which specific claims on performance are met. For video decoding, claims like MPEG-4 decoding are made, though without specifying the frame size or frame rate. The same holds for power figures, sometimes the power consumption of only the processing core is given. The power figures in Table 12 are the typical power consumption for the complete chip.

We will shortly discuss the most important features of the chipset in Table 12. Some of the ICs in Table 12 do not have a chip type yet, they are named with their 'codename'.

The three chip from Philips all contain one or more 32 bit TriMedia VLIW processors. In the PNX1500 one TriMedia is the only programmable processing unit available. It has a variable length decoder to support MPEG-2 elementary stream demultiplexing. Furthermore it support the direct connection of a LCD screen.
Table 12: Summary of the main features of several chipsets.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>IC type</th>
<th>MPEG-2 Advanced Simple Profile / VGA @ 30 fps</th>
<th>Direct LCD connection</th>
<th>External SDRAM Memory [MB]</th>
<th>External bus type</th>
<th>Processor type(s)</th>
<th>Hardware accelerators</th>
<th>Power Consumption typical [Watt]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Philips</td>
<td>PNX1500</td>
<td>√</td>
<td>√</td>
<td>8-256</td>
<td>PCI+XIO</td>
<td>TM3260 VLIW / 300 MHz</td>
<td>VLD</td>
<td>1.5</td>
</tr>
<tr>
<td>Philips</td>
<td>PNX8526 (Viper1.1)</td>
<td>3x</td>
<td>√</td>
<td>-</td>
<td>PCI+XIO</td>
<td>MIPS3940 / 150 MHz</td>
<td>VLD MPEG-2</td>
<td>2.5</td>
</tr>
<tr>
<td>Philips</td>
<td>PNX8550 (Viper2)</td>
<td>3x</td>
<td>√</td>
<td>16/32/64/128</td>
<td>PCI+XIO</td>
<td>MIPS4450 / 286 MHz</td>
<td>VLD MPEG-2</td>
<td>4.5</td>
</tr>
<tr>
<td>Intel</td>
<td>Bulverde+ Marathon4</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>XScale / 550 MHz</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Sigma</td>
<td>designs EM847x</td>
<td>√</td>
<td>√</td>
<td>2/4</td>
<td>PCI</td>
<td></td>
<td>√</td>
<td>1.2</td>
</tr>
<tr>
<td>Texas</td>
<td>Instruments TMS320DM270</td>
<td>√</td>
<td>√</td>
<td>32-128</td>
<td>C54x DSP / 100 MHz</td>
<td>VLD/ VLC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Texas</td>
<td>Instruments TMS320DM642</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>C64 VLIW / 600 MHz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Texas</td>
<td>Instruments TMS320DSC24</td>
<td>√</td>
<td>√</td>
<td>16</td>
<td>TMS320C5000 DSP</td>
<td>VLD/ VLC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

√ = supported
- = not supported

VLD = variable length decoder
VLC = variable length coder

3 Some vendors claim to support MPEG-4 Advanced Simple Profile, other vendors claim to support VGA resolution with 30 frames per second. This column summarizes if one of both claims is made.

4 Bulverde and Marathon are the Intel codenames for their next generation chipset for the handheld market. Bulverde is the successor of the PXA255 (currently used in iPronto). Marathon is the add-on chip for multimedia acceleration.

5 Both chips, Bulverde and Marathon, are necessary to perform media decoding. Reported estimated load of the Bulverde chip is 91%.

6 Advanced Simple Profile Level 5 without global motion compensation.
The PNX8526 and PNX8500 have an second processor, a MIPS processor on board. Both chips also contain more dedicated images processing units, like scaling and/or compositing and a MPEG-2 decoder implemented as a hardware processor. In the PNX8500 two TriMedia processors are available to perform high quality media processing.

The TMS320DM270 processor is aimed at the camera market, though support MPEG-4 video. It can connect to an CCD chip and contains a C54 DSP processor and an ARM7 processor. The TMS320DM642 is capable of decoding multiple MPEG-2 video streams. For the Texas Instrument chip-sets no power information is available.

The Intel solution uses a dedicated, closed coupled, companion chip that connects to the Bulverde, the successor of the PXA255. The Bulverde and Marathon are closely coupled as the video decoding process is spread over both chip. The run length decoding and inverse quantization is done on the Bulverde, the inverse Zig Zag, inverse discrete cosine transform, motion compensation, colour space conversion and scaling is done on the Marathon chip. The accompanying load for VGA sized MPEG-4 is 91% on the Bulverde. This is based on preliminary simulation based data. As the Bulverde is the host processor in the system, this leaves little headroom for the complete iPronto software stack.

Furthermore it is worthwhile to note that several demonstration systems of portable tablets with these ICs exist. At the former Application Lab of Philips Component, an XScale based portable tablet named ‘Picasso’[22] is developed. It uses Linux as operating system and features a mini PCI bus with extension connectors for removable cards. Optional cards include the Sigma Design EM8574 and the PNX1300, a predecessor of the PNX1500.

A PNX8526 based portable tablet named ‘HOTMAN-2’[22] is being developed at the Advanced Systems Laboratory of Philips Semiconductors. The ARM processor in the PNX8526 runs the Linux operating system. The TriMedia processor is used to support the media processing functionality. It is designed to fit into the iPronto casing. The LCD screen, touch panel, USB interface, local buttons, battery, speakers and microphone are all re-used from the iPronto design.

The selection of a specific integrated circuit not only depends on the features of the chipset from Table 12. Other factors included in a decision are the availability and quality of tooling like software compilers, debuggers, the level of support available from the manufacturer, existing third party software libraries, reference designs and the price of the integrated circuit.

**Proposed hardware extensions**

Of the media accelerators in Table 12, only three chips, the PNX1500, EM847x and the TMS320DM642 satisfy the requirements of MPEG-2 and MPEG-4 ASP decoding and have a power consumption smaller than 2 Watt.
Of these chips, the TMS320DM642 has a reported price level of $85 in quantities of 1000. This high price level makes it uninteresting to apply.

The power usage of the PNX1500, estimated at 1.5 W, includes running all IP blocks in the design, not only the VLIW processor. The power consumption of the design will decrease when unused IP blocks in the PNX1500 are shut down. Next the PNX1500 can directly interface to the LCD screen. This is not possible with the EM847x. The PNX1500 provides more flexibility by its programmable core VLIW, the flexibility of the EM847x is unknown.

This leads to the selection of the PNX1500 as the IC to extend the current iPronto design. In combination with the SA5250 and SA5251 wireless chipsets, this leads to the hardware configuration in Figure 37.

![Figure 37: Proposed hardware extension](image-url)
6. Conclusion and recommendations

6.1. Conclusions
This report shows how to extend a wireless tablet for streaming multimedia. The concepts described take the hardware and software of the system into account.

A top-level architecture is introduced that consist of two separate domains: A streaming and a control domain. The streaming domain handles all signal processing related computations. The control domain handles all event driven processing, like the user interface application and middleware software of the system.

The signal processing functionality of the streaming domain is modelled as a concurrent process network. For the hardware of the streaming domain a template is assumed consisting of processing units with an interconnect structure. The interconnect structure can give guarantees on throughput between processing units.

A (set of) task(s) of the process network is represented in the control domain by a software component. A component has defined interfaces that can be classified into two parts: a set of pins representing the in- and outputs for the signal processing functionality, and a set of component specific control interfaces for setting and getting signal processing parameters. A task can have mappings on different processing units. A component has references to all these mappings. These references are annotated with cost figures.

A grouping concept for components called services is proposed. A service is a software entity that provides a meaningful and coherent activity, a building block to the application. Services can interact with components by using a graph manager. The graph manager introduces a single point of interaction for applications to control the processing functionality of the streaming domain.

The graph manager uses a resource manager to obtain a mapping of the tasks on the processing units. The resource manager uses the cost figures for the creation of the best possible mapping of the tasks in the streaming domain. Next the accompanying point to point connections between processing units are given by the resource manager.

6.2. Recommendations

iPronto extensions
We advice the usage of the MultiMedia API (MMAPI) as high level abstraction for media processing. The MMAPI has a small set of mandatory software interfaces and is lightweight in memory footprint. Next, the introduction of streaming services is advised. These services model the streaming functional-
ity to the MMAPI. The MMAPI provides playback of content from a description in the form of a URI, and supports several popular network protocols like HTTP and RTP.

For the extension of the hardware design of the iPronto, with current available hardware, we propose to extend the design with an PCI bus. The PCI bus can be used for connecting a different wireless network card, and a media processing IC. For the wireless network the SA5250 and SA5251 ICs are proposed, supporting the IEEE802.11a/b/g standards. For the media processing the PNX1500 media processor is proposed, providing processing capabilities for MPEG-2 main level at main profile decoding and MPEG-4 Advanced Simple decoding.

Furthermore we advice to track the capabilities and price levels of FPGAs. The flexibility, price level and amount of processing offered are approaching interesting levels for consumer electronic devices.

Further Research

On an end-to-end system level, from the device providing the streaming content, to the device 'consuming' the content, the best packet size and the network injection should be investigated. Devices between the source and the sink, like a wireless access point, provide buffering for data. The IEEE802.11 protocol allows clients of wireless access points to periodically wake-up to receive buffered data. This allows the power consumption of the wireless network card to be reduced, as this allows for power down in the intermediate time.

An interesting option for the abstraction of the processing functionality of the streaming hardware is the usage of the virtual filesystem under Linux. The usage of filedescriptors and file based I/O could be an simple and easy to use interaction with the streaming hardware. Further investigation is necessary to clarify if this a viable path.
References

[1] OSGi Service Platform, release 2
   October 2001, Open Services Gateway initiative
   http://www.osgi.org

   Intel Corporation, March 2003
   doc. id 278693-001

   Advanced Linux Programming
   New Riders Publishing, Boston, 2001, USA
   ISBN 0-7357-1043-0

   The Java Language Specification
   MA, June 2000
   ISBN 0-201-31008-2

   Intent Manuals
   Tao Group Ltd.

   ej-technologies GmbH
   http://www.ej-technologies.com/products/jprofiler/overview.html

   Microsoft Windows 2000 Resource Kit
   Microsoft Corporation
   http://www.microsoft.com/windows2000/techinfo/reskit/

[8] D. Anderson
   PCMCIA System Architecture: 16 bits card
   MindShare Inc., 1995
   Addison-Wesley
   ISBN 0-201-40991-7

   A simple and efficient bus management scheme that supports continuous streams.
   ACM Transactions on Computer Systems, Volume 13, Issue 2 (May 1995), Pages: 122 - 140
   ACM Press, New York, NY, USA, 1995
   ISSN:0734-2071

    DVP-API2: Rationale & Architectural Highlights
    Proceedings of the 7th Philips Software Conference
    Eindhoven, june 2002
[11] *Java Media Framework API guide, JMF 2.0 FCS*
   November 1999
   Sun Microsystems

[12] *Mobile Media API (JSR-135) Specification*
    Version 1.0, 24 June 2002
    [http://www.jcp.org/aboutJava/communityprocess/final/jsr135/](http://www.jcp.org/aboutJava/communityprocess/final/jsr135/)

[13] Java Community Process
    [www.jcp.org](http://www.jcp.org)

[14] Gamma E., R. Helm, R. Johnson and J. Vlissides
    *Design Patterns, Elements of Reusable Object-Oriented Software*
    Addison-Wesley, 1995

    *RFC2396: Uniform Resource Identifiers (URI): Generic Syntax*
    Internet Engineering Task Force, 1998

[16] Nikolova M. and H. Cai
    *WWICE 2 Graph Mapper Architecture*
    Philips Research Technical note 2002/133

[17] *SA5250; SA5251 Product brochure*
    Philips Semiconductors

[18] A. Lewis
    *MPEG-4 over wireless networks*
    White paper
    Arm Ltd., Cambridge, United Kingdom, 2000

[19] Xilinx Corporation
    *Virtex-II Pro Platform FPGAs: Introduction and overview*
    Document DS083-1, March, 2003

[20] *MPEG-2 video decoder MP@ML for FPGA*
    Amphion CS6651
    2002, Amphion Semiconductor Ltd.

[21] *Picasso Mobile Multimedia Player project*
    Personal communication with R. Proveniers.

[22] Hendricksen R.H.M.
    *‘HOTMAN-2’ User Guide*
    Advanced Systems Laboratories, Philips Semiconductors, November 2003
    [http://pww.stce.sc.philips.com/hotman/](http://pww.stce.sc.philips.com/hotman/)
A Measurement CPU load Web browser at start-up on iPronto.

The measurements on the iPronto emulator show a large CPU load with a long duration at the first start-up of the web browser application. To verify this behavior on the iPronto device itself, instead of using the emulator to profile the software, the CPU load at the start-up of the web browser application on the iPronto was measured using the Cyclesoak [A] program.

The cyclesoak program is described by its author as:

*Cyclesoak is a tool for measuring system resource utilisation (CPU cycles and memory bandwidth). It uses a 'subtractive' algorithm: it measures how much system capacity is still available, rather than how much is consumed. This gives a very accurate and useful measurement of the efficiency of kernel subsystems. Much better than conventional process accounting, which can easily be wrong by an order of magnitude.*

A background cycle-soaking task is executed on all CPUs and 'cyclesoak' measures how much the throughput of the background tasks is degraded by other tasks on the system. This means that ALL effects of activity are measured - interrupt load, softirq handling, memory bandwidth usage, etc. This is much more accurate than using Linux process accounting.

The cyclesoak program was cross-compiled for XScale. Next a 'calibration' run on an empty iPronto was done. This was achieved by killing all elate processes on iPronto from the debug prompt. (The iPronto has serial debug interface that enables a login as root on the Linux OS). Killing all elate processes removes the complete application stack: the Java virtual machine and all upper software layer like OSGi, the service bundles and the applications.

Next a measurement of the CPU load at browser start-up was done. The iPronto was rebooted to create a clean system with a correct running software stack. The measured CPU load is given in the next figure:

![CPU load graph](image)

The web browser application is started at $t = 18$. The CPU load ends at $t = 169$ second. Thus in total the web browser uses 151 seconds to complete
load, including the retrieval and display of the 'homepage'. This is approximately 10 times the duration of same measurement in the iPronto emulator on the desktop PC.

[A] Cyclesoak homepage: