RWTH AACHEN UNIVERSITY

iStuffMobile: Rapidly Prototyping Novel Interactions for Mobile Phones

Master Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Faraz Ahmed Memon

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Stefan Kowalewski

Registration date: 7th Dec , 2005
Submission date: 17th May, 2006

Contents

List of Figures

List of Tables

Abstract

iStuff Mobile is a rapidly prototyping toolkit which is targeted towards mobile phone interaction designers. iStuff Mobile helps interaction designers explore novel interactions including sensor enhanced mobile phone interactions and ubiquitous computing interactions in which mobile phone acts as an input or output device to the environment. iStuff Mobile enables interaction designers to quickly prototype and test novel mobile phone interactions without making internal hardware/software modifications to the handset. iStuff Mobile leverages iStuff Framework [Ballagas et al., 2003] to provide a sensor network platform (Smart-Its), a mobile phone software and related patches for Apple Quartz Composer [Apple Quartz Composer].

This Thesis starts by presenting a comprehensive introduction of iStuff Toolkit and Smart-Its technology [Beigl et al., 2003b]. It continues by describing Related work done in this area. The idea of iStuff Mobile is then put forward along with the architectural description and example interactions prototyped using this toolkit. Finally the thesis concludes with a notion of future work.

Acknowledgements

This Master Thesis has been produced by technical and moral support of many individuals whom I would like to thank. Special thanks go to people mentioned below.

First of all I would like to thank Prof. Jan Borchers for making an exception and allowing me to do master thesis at his department because only those students are selected to do thesis at his department who took a few courses with him during their studies. Tico Ballagas, my thesis supervisor has been very supportive throughout my thesis. He was always available for discussions and gave me time under busy situations where anyone would have excused. Whether it was programming, documentation or resource allocation Tico was always there to help. So I thank Tico for everything that he has done to help me during this thesis.

People who supported me every now and then include my friends and colleagues. Eugen Yu helped me at places with Java code, Fahad Bin Tariq and Mohammad Sajjad helped me explore new ideas, Thorsten Kareer provided such a wonderful template for thesis write up, Junaid Ansari helped me setup the Latex environment for documentation and Imran Ahmed Khan helped reviewing this thesis. I particularly got support from Philip at the Teco forum[a] regarding the Smart-Its technology.

Last but not least, I would mention my parents and my fiancee Faryal Gul Memon for morally supporting me during frustrating times, for comforting me and for being there for me whenever I needed them.

Thank you!

[a]http://forum.teco.edu/

Conventions

Throughout this thesis we use the following conventions.

Text conventions

Definitions of technical terms or short excursus are set off in coloured boxes.

EXCURSUS:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition:
Excursus

Source code and implementation symbols are written in typewriter-style text.

`myClass`

The whole thesis is written in Canadian English.

Download links are set off in coloured boxes.

File: myFile[a]

[a]http://media.informatik.rwth-aachen.de/~ACCOUNT/thesis/folder/file_number.file

Chapter 1

Introduction

"I begin by taking. I shall find scholars later to demonstrate my perfect right."

—Frederick (II) the Great

Mobile phones are packed with extensive number of features these days

Mobile phones have become a part of daily life around the globe. Today, people use mobile phones not only for making/receiving calls but also for taking pictures, listening music, organizing tasks, sending/receiving emails etc. More and more features are being packed into mobile phones.

New mobile interactions cost time and money

Introducing sensors and actuators in mobile phones enables a variety of new interaction techniques. Interaction designers need to rapidly produce running prototypes to test this concept with users. However, hardware limitations of the mobile phones may hinder the ability to explore these novel interactions in a cost-effective manner. The number of iterations in design process of mobile interactions can be increased by lowering the time and financial cost of prototyping, which in turn would increase the quality of the user interface design [Nielsen, 1993].

iStuff Mobile enables interaction designers to design new interactions on

iStuff Mobile [Ballagas et al., 2006a] [Ballagas et al., 2006b] is a toolkit which facilitates mobile phone interaction designers to prototype new interactions using the mobile phones that are readily available in the market, saving them readily available mobile phones

time, cost and effort. iStuff Mobile extends the iStuff toolkit [Ballagas et al., 2003] to support rapid prototyping of novel interactions. It allows interaction designers to augment mobile phone with externally attached hardware such as Smart-Its [Beigl et al., 2003b] sensor network module. Using iStuff Mobile, interaction designers can prototype sensor driven mobile phone interactions. Additionally, the toolkit can be used to prototype ubiquitous computing applications such as mobile phone interactions with interactive spaces [Pering et al., 2005] or public displays [Ballagas et al., 2005]. Following are the contributions of the this thesis:

- An application that runs as a background service on the mobile phone to expose critical functions, allowing remote control of the foreground applications on the phone.
- A new graphical front-end for Smart-Its that simplifies configuration of the sensor nodes and integrates Smart-Its into iStuff framework through a proxy strategy proxy strategy.

A comprehensive introduction to iStuff toolkit and Smart-Its technology follows:

1.1 iStuff Toolkit

Definition: *Post-Desktop*

POST-DESKTOP:
Generally, post-desktop means beyond the desktop. Specifically, it means interaction with computers when they are embedded into walls, tables, chairs etc. When there may be no mouse or keyboard present for interaction. Ubiquitous computing is a subset of post-desktop computing

iStuff toolkit allows exploration and design of novel interactions in a multiuser, post-desktop environment where several applications collaborate with each other

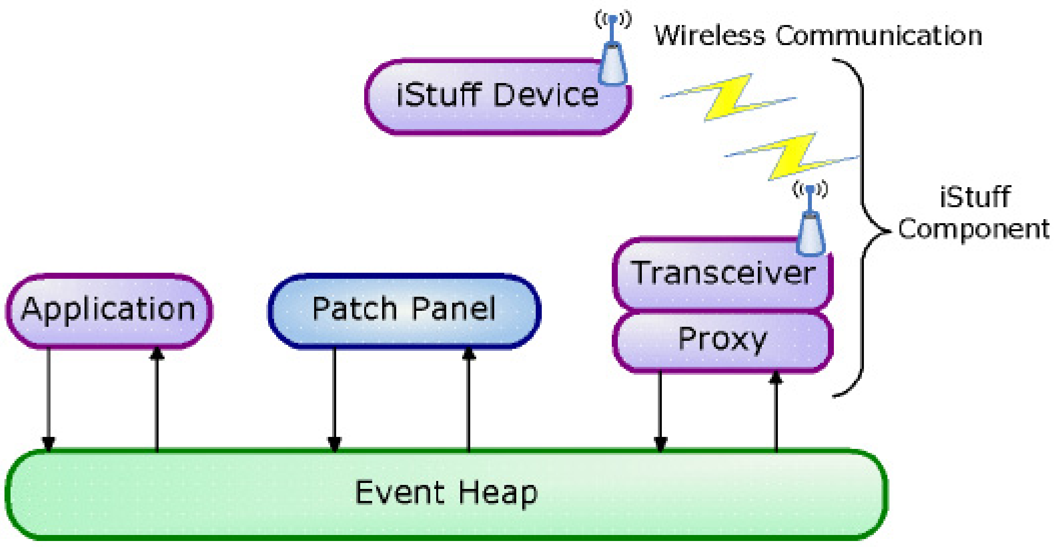


Figure 1.1: iStuff Architectural Diagram

[Ballagas et al., 2003]. iStuff toolkit is a freeware and could be downloaded from iStuff Project website[1]

1.1.1 iStuff Architecture

An Event Heap, iStuff Components and a Patch Panel makes up the iStuff toolkit

iStuff toolkit is based on the Event Heap [Johanson and Fox, 2002] which is the center of communication between devices and application in an interactive space. The toolkit also includes iStuff Components that are physical user input/output devices with wireless communication capabilities, and a Patch Panel [Ballagas et al., 2004] which dynamically maps events from input device to the events required by output device. Figure 1.1 shows the architecture of iStuff toolkit.

[1]http://media.informatik.rwth-aachen.de/istuff/

1.1.2 Event Heap

Definition:
Interactive Workspace

INTERACTIVE WORKSPACE:
An interactive workspace is a localized ubiquitous computing environment where people come together for collaboration [Johanson et al., 2002a]

An Event Heap is a coordination model similar to tuplespaces which allows diverse applications/devices to collaborate in solving problems in an interactive workspace.

As explained in [Johanson and Fox, 2002], in a tuplespace coordination between all the participants takes place through a shared space. A tuplespace allows tuples which are collection of ordered type-value fields to be posted to the the space, or read from the space in a destructive or a non-destructive manner. A retrieving application specifies the tuple which it wishes to retrieve from the tuplespace through a template tuple. A template contains precise values of fields to be matched, and wild cards for fields containing data to be retrieved. Following are features of the Event Heap (for details of these feature please refer to [Johanson and Fox, 2002]):

- **Anonymous Communication:** Communication between two applications is automatic as long as they both understand the same event types.
- **Interposability:** An intermediary can be used to transform an event that a source generated to an event that a receiver expected.
- **Snooping:** Snooping an event is possible in the tuplespace model. Snooping an event does not hinder the behavior of the receiving application.
- **Tuple Sequencing:** Events are sequenced in the order they are posted. The receiver will get the earliest matching event requested.
- **Expiration of Tuples:** A ”TimeToLive” field is provided in every event. TTL field specifies how long

an event will stay in the Event Heap before being destroyed.

- **Default Routing Fields:** Few default fields are added to an event by the Event Heap to ensure the correct routing of events.

- **Query Registration:** Applications can register to receive template tuples from the Event Heap. Whenever a matching event is placed onto Event Heap, a callback function of the applications registered for that event is called.

1.1.3 iStuff Components

TRANSCEIVER:
A transceiver in iStuff context, is a part of iStuff Component which does transmission (and reception) of data to (and from) an iStuff Device. See Fig 1.1

Definition:
Transceiver

PROXY:
A proxy in iStuff context, encapsulates the data transmitted from the iStuff Device into events (or extracts data from events to be transmitted to the iStuff Device). See Fig 1.1

Definition:
Proxy

iStuff Components are physical input/output devices with corresponding transceivers and proxies

iStuff Components are physical devices which are paired to a Transceiver and corresponding Proxy. The Transceiver + Proxy are in turn connected to the Event Heap. An iStuff Component must contain both device and a Proxy to connect to the Event Heap although many devices can share the same Proxy [Ballagas et al., 2003]. iStuff Devices contain simple input/output devices like buttons, sliders and buzzers, as well as more complex devices like Smart-Its and mobile phones. Figure 1.2 shows some of the iStuff Components.

To make sure that an iStuff Device is independent of wireless protocol or technology, the iStuff Proxy encapsulates the data transmitted from the iStuff Device into events (or

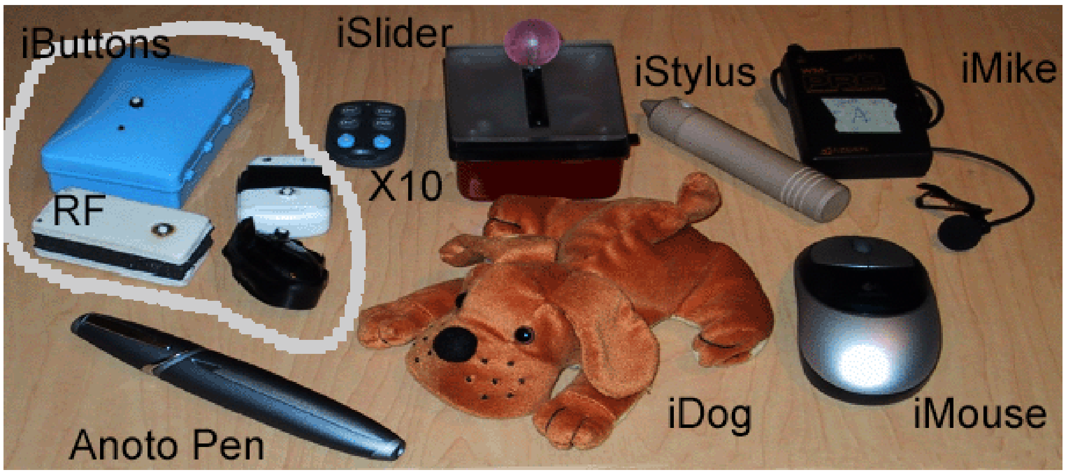


Figure 1.2: Example of iStuff Components [Ballagas et al., 2003]

extracts data from events to be transmitted to the iStuff Device). These events are posted onto (or received from) the Event Heap.

1.1.4 Event

Event in the iStuff toolkit is a message or tuple which consists of a type field and any number of optional fields in form of key-value pairs [Ballagas et al., 2003]. These events are basis of communication between applications and devices. Event Heap is the channel through which events are transmitted.

1.1.5 Event Communication

iStuff Components communicate with each other through events which are transmitted over the Event Heap

The communication between iStuff components and applications takes place through events. An iStuff component posts events onto the Event Heap and any application can register to receive these events from the Event Heap. The registration takes place by specifying event type, and optionally other criteria based on matching of specific fields [Ballagas et al., 2003].This communication mechanism allows several application and devices running on separate machines to communicate in an interactive manner.

An iStuff component may produce events which are specific to its properties e.g. An "iMouse" may produce an event of type *NewMousePosition* but an application which would register for such event would become specific to "iMouse". A better and more flexible way would be to introduce a level of abstraction and expect e.g. *NewPosition* event at the application end. This way an application would be able to handle the new position of "iMouse", "iSlider", "iTouchPanel" etc. Patch Panel would then be used to dynamically transform an event of type *NewMousePosition* into an event of type *NewPosition*.

1.1.6 Patch Panel

In a ubiquitous computing environment new devices/services may be added frequently which makes it clear that this environment will be incrementally deployed. The devices, application and services that already exist in such environment cannot anticipate communication with every other component they may encounter. They need to be able to communicate without a prior knowledge of each other and this communication should be meaningful to both the components as well as the user of the system. This phenomenon is known as the incremental integration.

Patch Panel enables dynamic mapping of events

The Patch Panel [Ballagas et al., 2004] provides a general facility for retargeting event flow. Hence, allowing incremental integration of devices into ubiquitous computing environments which use Event Heap as a communication space. In the Patch Panel intermediations can be expressed as simple event transitions or more complex finite state machines. The Patch Panel works by subscribing to all event types. On receiving an event which matches a trigger condition, the Patch Panel generates the corresponding output events and posts them onto the Event Heap. Here we will consider an example of simple event translation. For a better understanding of finite state mechanism for intermediation refer to [Ballagas et al., 2004].

We will use the same convention as used in [Ballagas et al., 2004] for describing the mapping. A mapping (*trigger* $\rightarrow$

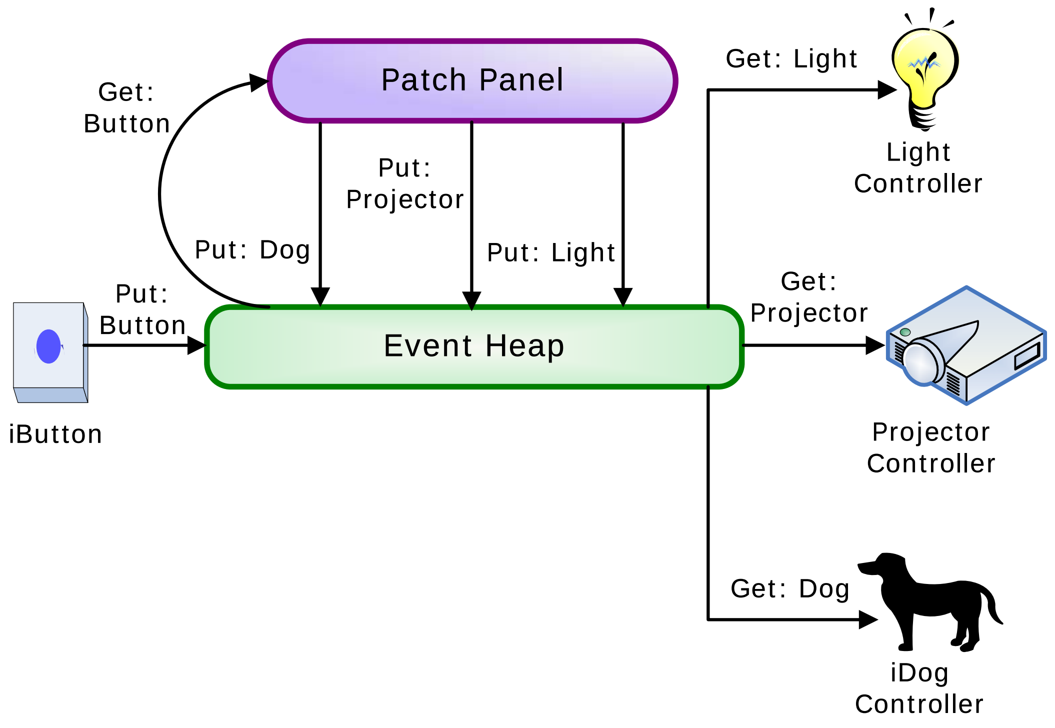


Figure 1.3: This figure shows an example of Patch Panel mapping. The iButton in this figure represents both the button and its proxy. Light Controller, Projector Controller and iDog Controller are proxies to Light, Projector and iDog respectively

output events) is the basic functionality provided by the Patch Panel. Lets say e.g. we have an "iButton" which whenever pressed generates an event of type *Button* with string-valued field `id`, a "Light Controller" which responds to an event of type *Light* containing an integer-valued field `brightness` with intensity values between 0 and 10, a "Projector Controller" which responds to an event of type *Projector* with boolean-valued field `powerOn`, and an "iDog Controller" which responds to an event of type *Dog* with a string valued field `action`. We would like to configure the Patch Panel in such a manner that whenever the "iButton" is pressed, the light goes on, the projector is turned on and the iDog barks. The following mappings can be specified in the Patch Panel:

Button(`id = blue`) $\rightarrow$ *Light*(`brightness = 10`),*Projector*(`powerOn = true`),*Dog*(`action = bark`)

Now whenever the iButton is pressed, the following course of actions take place: (Figure 1.3 depicts the idea)

- An event *Button*(`id = blue`) is posted onto the Event Heap.
- None of the "Light Controller", "Projector Controller" and "iDog Controller" recognizes the *Button* event but the Patch Panel recognizes it as a trigger for the Button mapping.
- The mapping shown above fires and the Patch Panel posts events *Lights*(`brightness = 10`), *Projector*(`powerOn = true`) and *iDog*(`action = bark`) onto the Event Heap.
- The "Light Controller" recognizes the *Light* event and turns on the Light with full intensity, "Projector Controller" recognizes the *Projector* event and turns the projector on and the "iDog Controller" recognizes *Dog* event and makes the iDog bark.

The example translation shown above is a simple one. Patch Panel allows more complex event translations, in which for example, the output event values may be derived through some mathematical calculations or conditions over input event values.

1.2 Smart-Its Technology

Smart-Its are small scale embedded systems with basic sensing, processing and actuating capabilities

Smart-Its also knows as particle computers are small scale low powered embedded systems. Attaching Smart-Its to every day objects adds basic computation, sensing, communication and actuating mechanism to that object [Beigl et al., 2003b]. All this functionality has been combined to

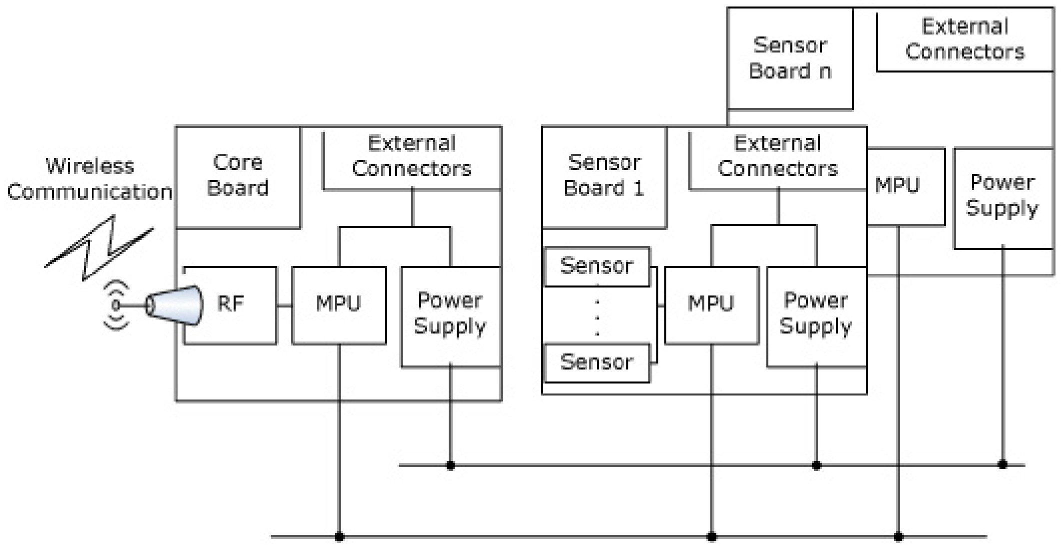


Figure 1.4: This Figure has been reproduced from the original in [Beigl et al., 2003b]. The figure depicts architecture of the Smart-Its platform

produce Smart-Its boards that can be used without any further infrastructure requirements. Just like paper Post-Its add information to some object, Smart-Its add basic computing, sensing, communication and actuating capabilities to an object.

A Smart-It is composed of two independent circuit boards

Smart-Its technology is based upon two independent circuit boards, namely a Core board and a Sensor board. The Smart-Its core board is mainly responsible for communication and processing of data while the sensor board consists of several sensors to gather the contextual information.

1.2.1 Smart-Its Architecture

As described in [Beigl et al., 2003b] Smart-Its architecture is driven by separation of communication, processing and filtering of data from processing and storage of sensor/actuator related information. To facilitate this functional separation, two independent hardware boards have been produced. Each of these hardware boards have their own processor, memory as well as system software.

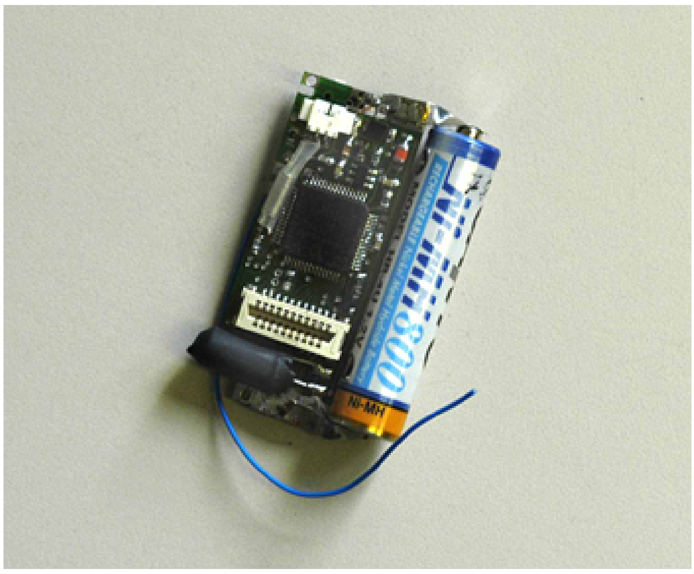
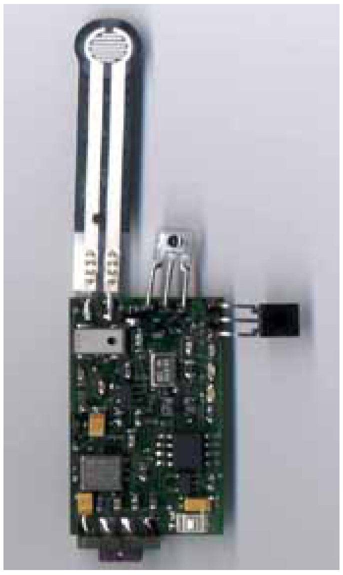



Figure 1.5: Left side shows Smart-Its core board and the right side shows Smart-Its Sensor Board

The functional separation in the Smart-Its architecture adds flexibility to the architecture by allowing developers to replace sensor boards with more sophisticated boards. These boards could be third party produced boards or ones produced by the developers themselves. The only requirement for such boards is to have a compatible interfacing module with the Smart-Its core board.

Smart-Its architecture allows replacement of the sensor board with more sophisticated boards

Figure 1.4 depicts the architecture of the Smart-Its platform. A typical Smart-Its is composed of one core board which is an RF board for high-level wireless communication, and zero to a maximum of 16 sensor boards. The sensor boards are connected to the RF board through serial lines for control and data flow. The core board as well as the sensor boards have their own processor, power supply, program and data memory.

zero to sixteen sensor boards can be attached to a Smart-It core board

1.2.2 Smart-Its Core and Sensor Boards

Left side of the Figure 1.5 shows a Smart-Its core board which is also known as the Particle Computer. The core board is the center of communication. The protocol used for communication is AwareCon, details on how AwareCon works could be found in [Beigl et al., 2003a]. The Smart-Its hardware is commercially available at Particle Website[2] . The hardware details of the core board including the list of complete features could be found at the Teco website[3]

The Spart Smart-Its Spart sensor board sensor board is equipped with several sensors for gathering contextual information Spart sensor board

Smart-Its sensor board also known as the Spart sensor board consists or several sensors including: acceleration sensors (X/Y/Z axis), light sensor, temperature sensor, force sensor, ambient light sensor and volume sensor. Additional sensors can also be connected to the sensor board using connectors. The hardware details of the Spart sensor board including list of complete features could be found at the Teco website[4]

1.2.3 Smart-Its Communication

On a basic level Smart-Its communicate with one another as soon as they are in range of each other. No further infrastructure, hardware or configuration is needed. Further, Smart-Its can communicate with the environment when they come near an "X-Bridge". X-Bridge runs over ethernet and allows Smart-Its to connect to any internet enabled device. The basic functionality of X-Bridge is to enable devices (e.g. PCs, PDAs) in the network to connect to Smart-Its and receive sensor data from it. Smart-Its can also access an internet service using an X-Bridge (Shown in Figure 1.6).

X-Bridge allows Smart-Its to connect to any internet enabled device on a LAN

Communication takes place between Smart-Its and X-Bridge in the following manner:

- When X-Bridge receives a packet from a particle computer, it removes the RF headers from the packet, adds UDP headers to the packet and forwards it to the Ethernet network.
- When X-Bridge receives a packet from Ethernet network, it removes the UDP header from the packet,

[2]http://www.particle-computer.de/
[3]http://particle.teco.edu/documentation/content/particle.html
[4]http://particle.teco.edu/devices/index.html

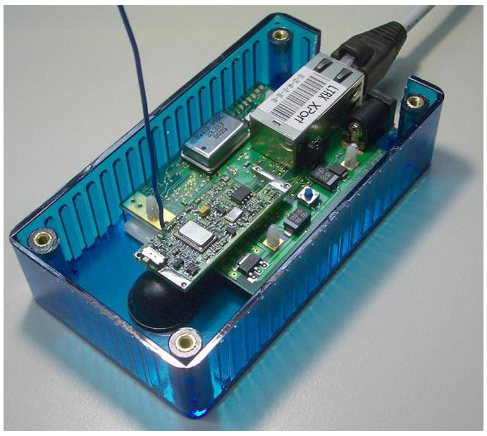


Figure 1.6: A Smart-Its X-Bridge

adds RF headers to the packet and forwads it to the destined particle computer.

- While forwarding packets from a particle computer to the network, X-Bridge adds the location information to the packets so that applications running on the network can locate particles.

Figure 1.7 shows how several particles communicate in a networked environment. The figure indicates that Smart-Its Communicate with an X-Bridge which is running over LAN, by sending RF packets containing the sensor information. The X-Bridge then converts the RF packets to UDP/IP packets and transmits them to the laptop which is running over the same network.

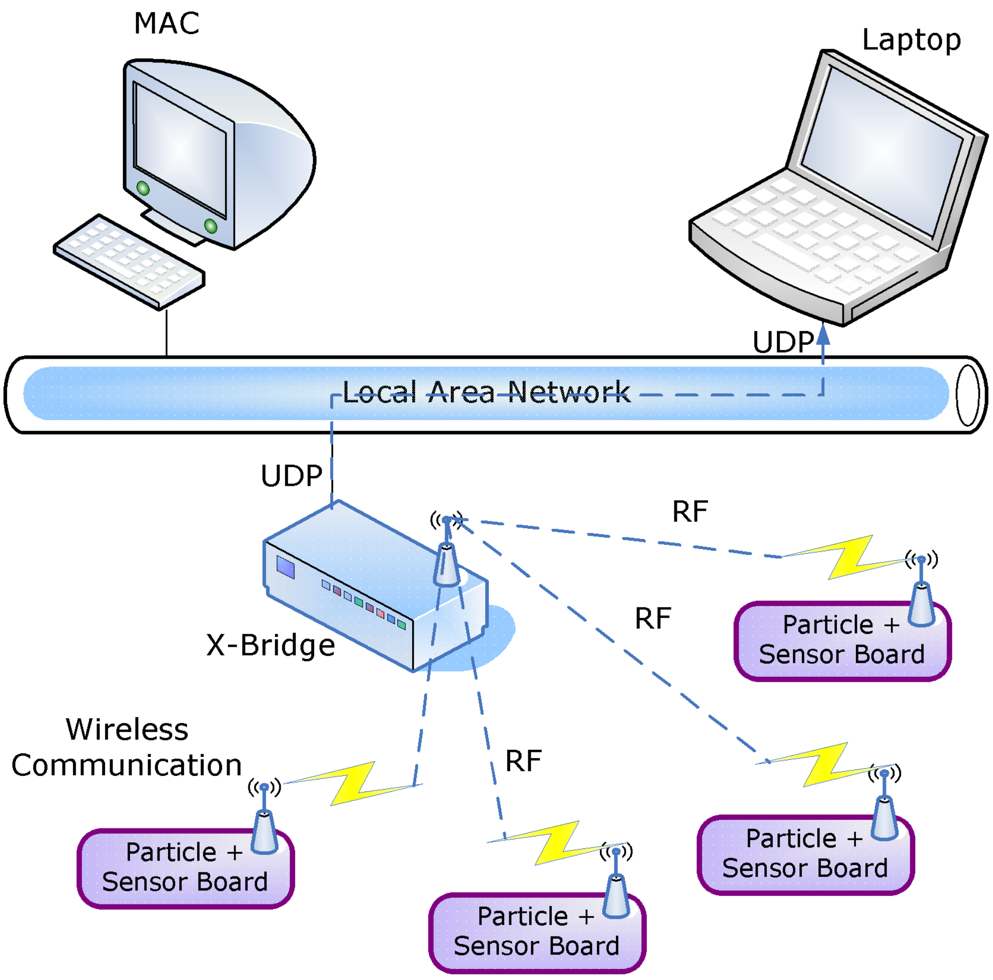


Figure 1.7: This Figure depicts Smart-Its Communication Network

Chapter 2

Related work

"The secret of success is to know something nobody else knows."

—Aristotle Onassis (1906-1975)

Several toolkits have been introduced in the recent years that help prototype interactions with physical devices. This section gives a brief introduction to these toolkits and discusses their similarities/differences with the iStuff Mobile toolkit.

2.1 Phidgets

Phidgets is a physical device prototyping toolkit which comes in with several hardware and software components

Physical widgets or more popularly known as Phidgets package input and output devices in a manner that their hardware level details are hidden from the end users. However, the functionality of these devices is made available through a well-defined API. Phidgets also provide a software connection manager to determine the connectivity of devices, a simulation environment and an optional on-screen component which helps interact with the devices [Greenberg and Fitchett, 2001].

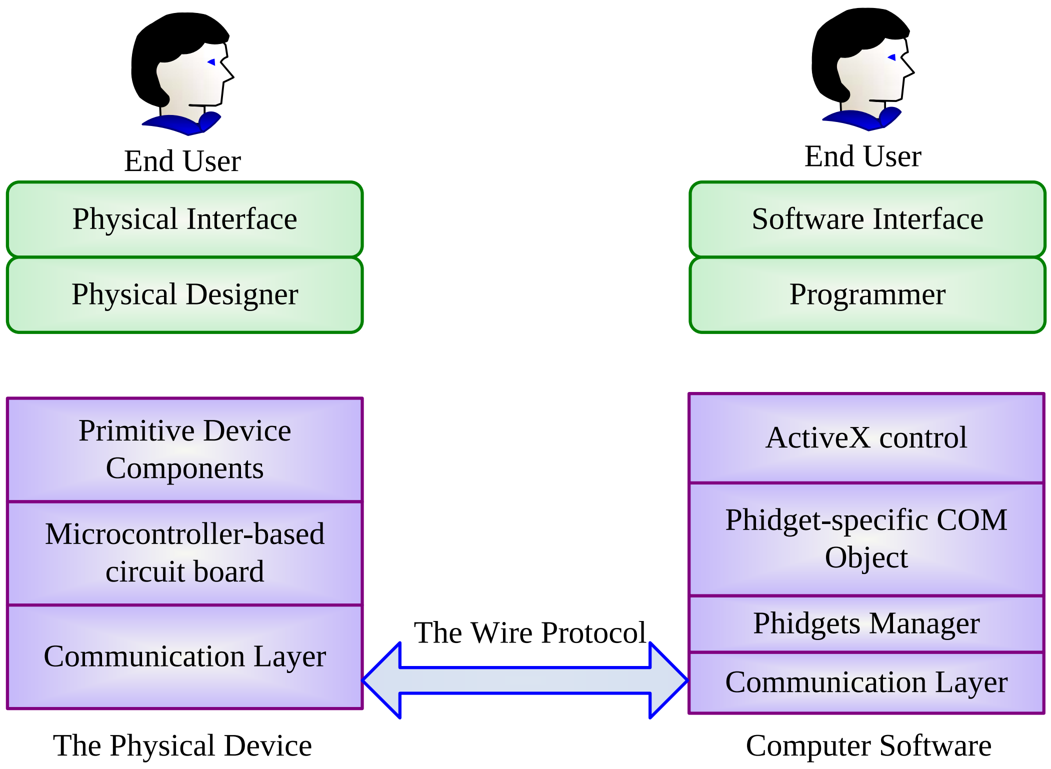


Figure 2.1: The Phidgets Architecture: This Figure has been reproduced from the original in [Greenberg and Fitchett, 2001]

2.1.1 Phidgets Architecture

Phidgets architecture is driven by several hardware and software components. Following is a short description of these components. For further details refer to [Greenberg and Fitchett, 2001]

Physical devices in Phidgets architecture include several I/O components and a communication network

The Physical Device: On the left side of Figure 2.1, one can see the physical device in the Phidgets architecture. The physical device includes different input/output components (e.g. buttons, sensors, motors etc), a micro-controller based circuit board and a communication layer. A Physical device is used by the interaction designer to create a physical interface which would be handed over to the end user. The communication layer is responsible for communicating with the host computer. The communication is based of a USB connection which is handled by the micro-controller.

The Wire Protocol: For a physical device to talk to a host computer the protocol used is known as the Wire Protocol (See Figure 2.1). The wire protocol is not visible to the programmers. It initiates the communication at a lower level. A software written for both micro-controller and the host computer (Windows 2000) is used to communicate using the wire protocol over a USB connection. Whenever a device is connected to the host PC, it appears as a USB device to the Windows 2000 and the communication is initiated.

The wire protocol initiates the communication between a physical devices and a computer in Phidgets architecture

The PhidgetManager: The PhidgetManager (See right side of Figure 2.1) is a COM object and this COM object generates events whenever a device is connected to, or disconnected from the computer. The PhidgetManager provides an API to end programmers through which they can detect all the devices attached to the computer.

The PhidgetManager is a COM object which handles device connectivity

Phidget-specific COM objects: Whenever a device is attached to the computer, the PhidgetManager creates a Phidget-specific COM object corresponding to that device. This COM object provides complete access to the physical device. The properties of the device can be accessed and changed at the runtime using the API provided by the Phidget-specific COM objects.

Phidget-specific COM objects provide access to individual devices and their properties

Phidget ActiveX controls: Phidget ActiveX controls are built on top of phidget-specific COM objects. These ActiveX controls provide COM objects with a visual interface. Programmers can use these ActiveX controls to simulate the physical devices, either they are connected or disconnected. They have a choice of using either these ActiveX controls or the phidget-specific COM objects directly.

Phidget ActiveX controls provide visual control of hardware components

2.1.2 Phidgets vs. iStuff Mobile

Although Phidgets enable prototyping of physical user interfaces, they do not provide explicit support for prototyping mobile phone interactions. A mobile phone is not available as a Phidget. However, iStuff Mobile enables prototyping of mobile phone interactions. The flexible architecture of iStuff framework allows phidgets to be used along with the iStuff Mobile to prototype more sophisticated mobile

No explicit support for mobile phones in Phidgets phone interactions.

Phidget devices are mostly wired unlike iStuff Mobile's complete wireless solution

Most of the Phidgets prototyping solutions are wired. However, iStuff Mobile provides a complete wireless solution. iStuff Mobile relies on Smart-Its [Beigl et al., 2003b] to provide a low-powered sensor board which communicates through a wireless network. This sensor board along with the Smart-It could be directly taped to the mobile phone. The mobile phone itself communicates with the proxy over bluetooth.

2.2 Calder

Calder uses small wired/wireless components to prototype physical devices

Calder is a hardware toolkit which is targeted towards interaction designers to help them in early design phases. The Calder toolkit enables prototyping of physical devices using small input and output components. These components can be either wired or wireless and are capable of communication with a computer [Lee et al., 2004].

2.2.1 Calder Architecture

This section summarizes the details of the architectural components of the Calder toolkit. For further details, refer to [Lee et al., 2004].

Calder wireless components communicate the PC through a wired uplink transceiver

Wireless Components: The Calder toolkit provides several wireless components which communicate with a computer through a wired uplink transceiver (See Figure 2.2). The wireless components communicate with the transceiver using conventional radio technology and the transceiver in turn is connected to the PC through a USB connection. These components are provided with a battery source and a microprocessor. To support attaching of the wireless components to a foam, two push-pins are provided at the back of each component.

Calder wired components

Wired Components: The wired components is the Calder toolkit include a general purpose input component and a

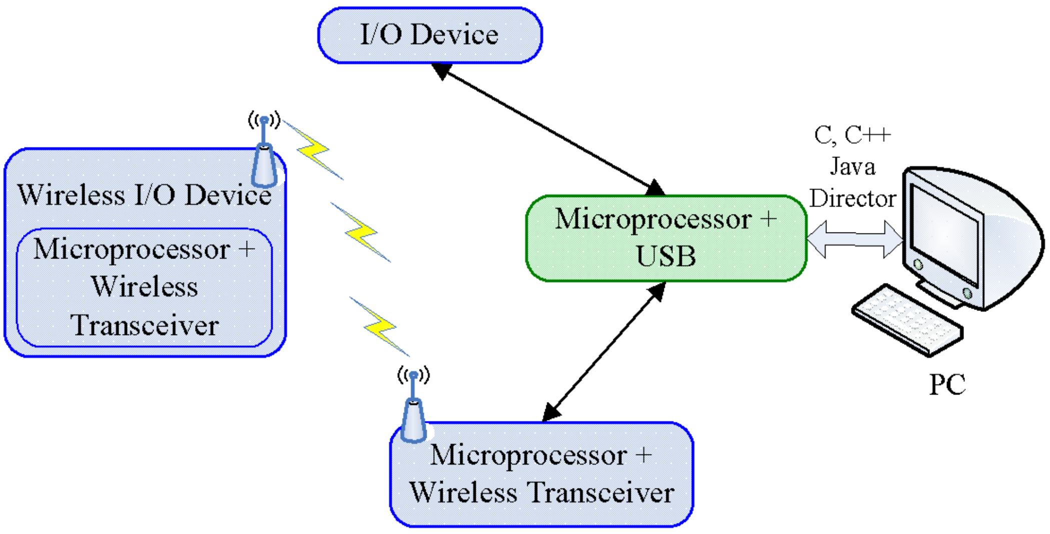

I/O Device
Wireless I/O Device
Microprocessor + Wireless Transceiver
Microprocessor + USB
C, C++
Java
Director
PC
Microprocessor + Wireless Transceiver

Figure 2.2: The Calder Toolkit Architecture: This Figure has been reproduced from the original in [Lee et al., 2004]

runtime configurable I/O breadboard. Using these components several input/output devices can be connected through a short cable. The wired components in the Calder toolkit communicate with the PC through a USB connection which is also the source of power for them (See Figure 2.2). These components are also provided with a microprocessor.

communicate with PC through a USB connection

Global Master (Computer): The computer in Calder toolkit architecture provides the processing power and a programming infrastructure. All the wired and wireless components are connected to the Global Master through a USB connection.

Programming Infrastructure: The Calder toolkit employs the same strategy as Phidgets for providing programming infrastructure. Each of the wireless or wired component in the Calder architecture is represented as an object inside a GUI system on PC. Change in the object state will invoke a change in the physical component and vice versa. Actions of physical devices which represent an input are fed into the computer as events. The physical devices can be accessed and manipulated using C access routines which are provided with the toolkit.

Wirless and wired components in Calder can be manipulated through GUI objects on PC

2.2.2 Calder vs. iStuff Mobile

Calder does not provide support for mobile phones

Prototyping mobile phone interactions is not possible using the Calder toolkit because it does not provide support for introducing a mobile phone as a Calder component. However, iStuff Mobile has been designed specifically for prototyping interactions using the mobile phone as input or output device to the environment.

Calder toolkit is targeted towards programmer rather than designers

To use the Calder toolkit a prior knowledge of object oriented programming in C++ is required in order to access the Calder components as objects in a GUI system. However, an interaction designer may not possess this knowledge. iStuff Mobile by extending the iStuff toolkit provides the Patch Panel GUI which eliminated the necessity of Object Oriented programming knowledge.

2.3 The TEA project

TEA project involved development of sensor boards to retrieve contextual information

The TEA (Technology for Enabling Awareness) Project is an effort to develop add-on components for context retrieval in mobile phones, communication devices, PDAs, laptops and GSMS. The TEA project developed two components: TEA-I board and the TEA-II board (See Figure 2.3). The TEA-I board is designed to communicate with a computer over a serial line. The board itself consists of 8 sensors to gather the contextual information. The TEA-II board has been developed so that it can fit into the large battery casing of the Nokia 6110-6150 series phones. The board consists of 8 sensors and two communication slots. One communication slot is a serial port to enable communication with PDAs and Computers and the other slot is a Nokia port for communication with Nokia 6110-6150 series phones. For further information on TEA boards, refer to [Technology for Enabling Awareness].

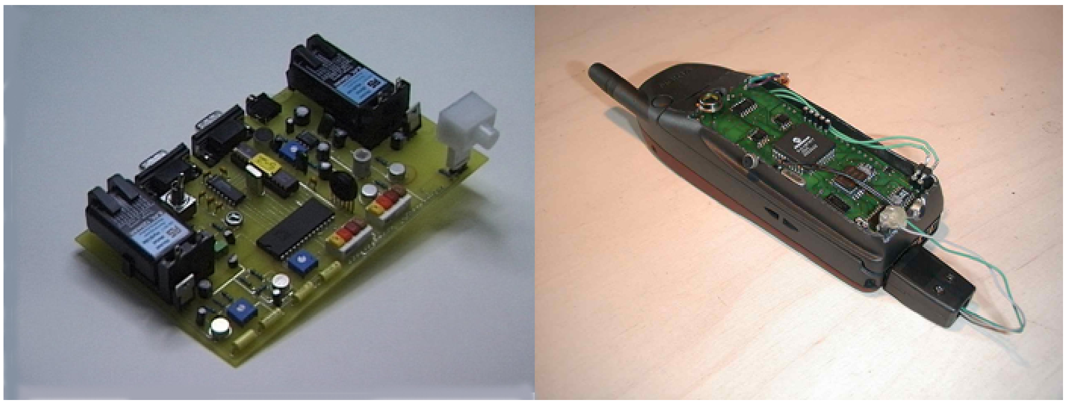


Figure 2.3: The TEA boards: Left side shows TEA-I board and the right side shows TEA-II board with the Nokia mobile phone (These images have been taken from [Technology for Enabling Awareness])

2.3.1 TEA Architecture

The TEA architecture is composed of four layers: sensors, cues, contexts and an application layer. The sensors are implemented as hardware while other layers are implemented as software layers [Schmidt et al., 1999]. Figure 2.4 depicts the TEA Architecture. A short description of layers in TEA architecture follows:

TEA architecture divides sensors into physical and logical sensors

Sensors: Sensors in TEA Architecture are divided into two categories: Physical sensors and logical sensors. Physical sensors are hardware components that measure the physical parameters in the environment. Logical sensors are information gathered from the host e.g. current time, IMSI number etc.

Cues provide abstraction of physical and logical sensors

Cues: A Cue in TEA architecture is function that that takes values of a single sensor up to a certain time as input and provides a symbolic or sub-symbolic output. Cue itself is an abstraction of physical and logical sensors. For physical sensors, cues also solve the problem of calibration [Schmidt et al., 1999].

Contexts depict current situation of a user/device

Contexts: The context in TEA architecture is determined by the available cues. Context depict the current situation of the user/device on an abstract level. The context in TEA is derived from a set of two-dimensional vectors. Each vec-

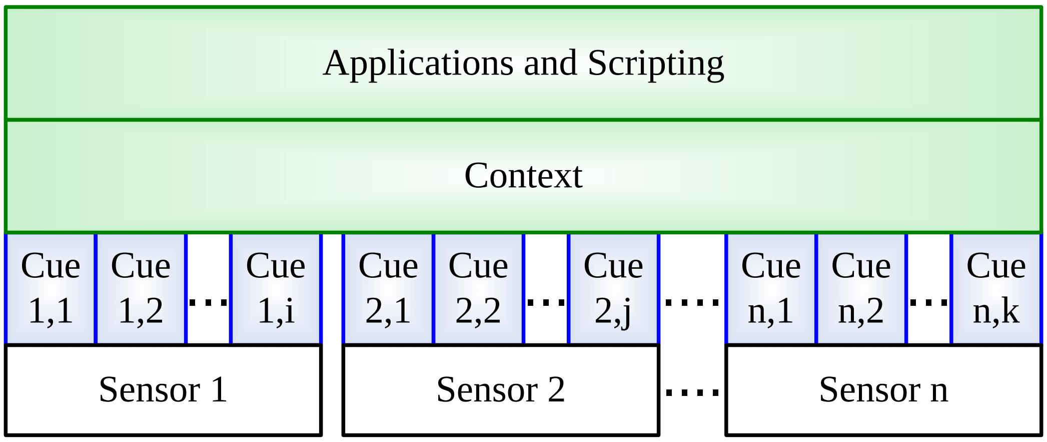


Figure 2.4: The TEA Architecture: This Figure has been reproduced from the original in [Schmidt et al., 1999]

tor consists of a symbolic value and a number determining the probability that the user/device is currently in this situation.

Scripting primitives provide access to contextual data through programming

Application and Scripting: The scripting primitives are offered to programmers in order to access the contextual information in application. Basic actions can be performed while entering a context, leaving a context and while in a certain context [Schmidt et al., 1999].

2.3.2 TEA vs. iStuff Mobile

TEA project only allows prototyping of interactions using sensors

The TEA project provides a limited amount of new interactions i.e. An interaction designer can use the TEA board only to prototype interactions which involve use of sensor data. In iStuff Mobile on the other hand, mobile phone can also be used as an input device to a ubiquitous computing environment. iStuff Mobile introduces a lose coupling between the sensor values and the application logic by allowing the relationship between the two to be modified at runtime. However, in order to modify the mapping of sensor values to application logic in TEA architecture, mobile phone application needs to be recompiled and redownloaded.

The TEA-I board is large sensor board which communicates with the computer over a serial line. The size of the board makes it hard to attach it to a mobile phone. iStuff Mobile provides a mechanism to use Smart-Its [Beigl et al., 2003b] sensor board which very small in size for accessing the contextual information. In iStuff Mobile the computer communicates with the mobile phone over a bluetooth wireless connection.

TEA-I board cannot be attached to a mobile phone

The TEA-II board has been developed to fit with into the large casing of Nokia 6110-6150 series mobile phones. However, iStuff Mobile covers Symbian series 60 [Symbian Series 60] mobile phones which is a wider range. The architecture of iStuff Mobile allows the toolkit to be theoretically used for all mobile phones.

iStuff Mobile covers a wider range of mobile phones than the TEA-II board

Finally, the TEA project is a complete wired solution while the iStuff Mobile is a complete wireless solution to prototyping mobile phone user interaction.

2.4 D.tools

D.tools is a prototyping toolkit that enables rapid prototyping of information appliances using the concept of integrated interaction. D.tools comes with several I/O hardware components that can be attached to a computer, and a PC-based visually authoring environment that works on the concept of state-charts. States in the visual authoring environment represent the device output while state transitions are fired by a physical input. The d.tools does the rapid mapping of graphical widgets to the physical I/O components [Klemmer et al., 2005].

D.tools comes with several I/O devices and a visually authoring environment for prototyping physical interactions

2.4.1 D.tools Architecture

D.tools is built up of a software and a hardware layer. Figure 2.5 depicts the architecture of d.tools. A brief discussion of the d.tools architecture follows. For further details refer to [Klemmer et al., 2005]

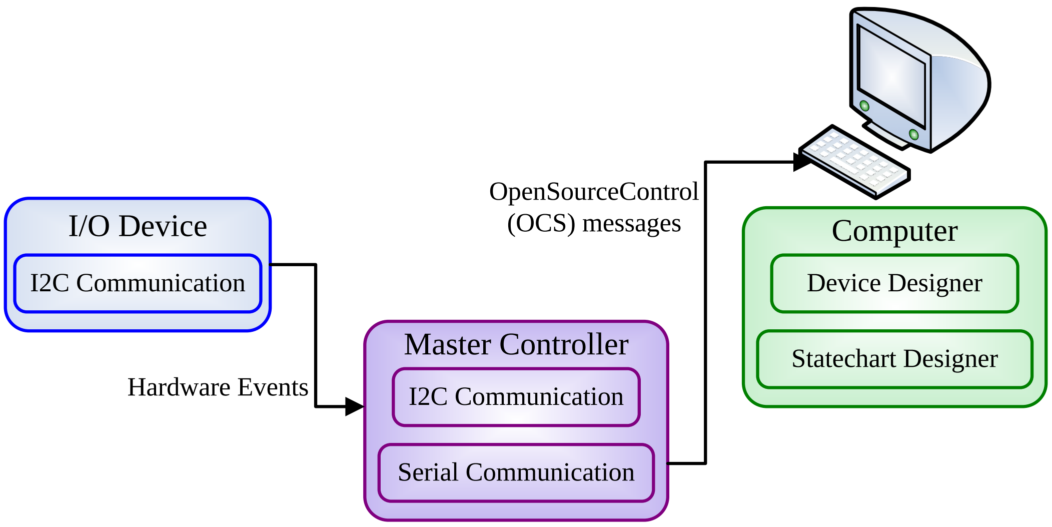


Figure 2.5: D.tools Architectural Diagram

D.tools hardware components are composed of a micro-controller and a communication module

Hardware Layer: Each hardware component in d.tools architecture consists of a micro-controller and a networking module to communicate over an I2C bus. A hardware component connects to a master controller board which co-ordinates the communication between the component and the PC. The master controller transforms the hardware events into OpenSourceControl (OCS) messages. The controller itself is connected to the PC through a serial line or a USB connection using virtual serial port driver.

D.tools provide a device designer and a statechard designer on the software level

Software Layer: The d.tools visual authoring environment consists of two main components: A device designer and a statechart designer. The visual authoring environment itself has been implemented as a plug-in for JAVA Eclipse IDE.

The Device Designer in the visual authoring tool allows designers to lay down a representation of the appliance they are prototyping. It allows arrangement and resizing of input/output components.

The Statechart Designer enables behavior prototyping of the appliance by creating interaction graphs. States in a startchart describe the values on the outputs at a particular instance in time while transitions represent control flow from one state to another.

2.4.2 D.tools vs iStuff Mobile

D.tools also provides a wired solution. iStuff Mobile on the other hand is a complete wireless solution. The wireless factor is important when dealing with physical sensors because it involves movement of the device in order to test it under different circumstances.

D.tools is a wired solution unlike complete wireless solution of iStuff Mobile

Designing new mobile phone interactions with d.tools would involve developing a physical mobile phone prototype with all the hardware components. However, iStuff Mobile allows the use of existing mobile phones for prototyping new interactions.

iStuff Mobile uses existing mobile phones to prototype new interactions unlike d.tools

The use of real mobile phones makes iStuff Mobile less flexible than d.tools for experimentation with form factor, but the scale of the prototypes is more realisitic. In addition, real devices afford more sophisticated UI design using the Nokia Series 60 [Nokia Series 60] software development kit (SDK) or Macromedia Flash Lite [Macromedia Flash Lite]. The iStuff graphical programming environment is not restricted to state machines, although the iStuff Framework does allow them 1.1.6—"Patch Panel". Finally, d.tools focuses on localized interactions, while iStuff Mobile also supports distributed interactions in ubicomp environments.

D.tools allows localized interactions unlike iStuff Mobile which allows distributed interactions

2.5 Teleo

Teleo is a commercial rapidly prototyping toolkit developed by MakingThings [Teleo].

2.5.1 Teleo Architecture

Teleo Architecture is less complex one than the ones described above. None the less, the architecture is built up of a hardware and a software layer.

Teleo hardware connects to PC through USB

Hardware Layer: Teleo comes with a number of I/O hardware components which can be connected to a computer using a normal USB connection.

Teleo hardware can be accessed from PC through programming languages

Software Layer: The Teleo hardware components connected to a computer can be accessed and programmed using a number of programming languages, mentioned specifically C++ and Macromedia Flash (MX 2004 and Action Script 2.0) [Macromedia Flash].

2.5.2 Teleo vs. iStuff Mobile

Teleo is targeted towards programmers

Teleo toolkit for rapidly prototyping physical UIs is targeted towards programmers rather then designers. A designer may not possess the programming knowledge of C++. iStuff Mobile on the other hand requires almost no programming knowledge to provide prototyping solution for mobile phones.

Teleo is a wired solution

Like most of the other toolkits Teleo uses wired connection. Further, teleo does not support mobile phone interaction design.

2.6 CCC Cybelius Maestro

CCC Cybelius Maestro is a commercial product developed by Cybelius that support design, simulation and code generation for telecommunication and electronic products [CCC Cybelius Maestro].

2.6.1 CCC Cybelius Maestro Architecture

CCC Cybelius Maestro is a software prototyping toolkit

CCC Cybelius Maestro is Software Prototyping toolkit which supports all major operating systems. It is an open architecture toolkit which allows prototyping, simulation and testing of products. The architecture is realized by a Connectivity Framework and a Plug-in Tool Framework. For further details refer to [CCC Cybelius Maestro].

Connectivity Framework: Connectivity Framework in CCC Cybelius Maestro architecture provides interfaces for integrating external tools, simulation models, software and hardware into the CCC Cybelius Maestro simulation environment.

Plug-in Tool Framework: Plug-in Tool Framework allows designers to build and add new tool components dynamically into the CCC Cybelius Maestro environment.

2.6.2 CCC Cybelius Maestro vs. iStuff Mobile

CCC Cybelius Maestro allows simulation of mobile phone interactions. However, iStuff Mobile prototypes interactions on a real mobile phone

CCC Cybelius Maestro is a Software Simulation environment which enables designers to prototype physical devices and simulate them in a software environment. CCC Cybelius Maestro also strongly supports simulation of mobile phone interaction but that is also limited in a software environment. iStuff Mobile however enables designers to prototype interactions on a real mobile phone which makes it more realistic for use. This factor of the iStuff Mobile is most important one because designers cannot get that realistic feel unless the simulation occurs on a real device.

The composition of hardware and software layer allows iStuff Mobile to enable exploration of physical interactions and interaction in an interactive space unlike CCC Cybelius Maestro.

2.7 ContextPhone

ContextPhone allows developers to use contextual information in applications that are

ContextPhone is a software platform that enables developers to incorporate contextual information into their application when developing for Symbian series 60 [Symbian Series 60] mobile phones. ContextPhone provides four interconnected module as a set of open source C++ libraries developed for smart phones

which can be used by the developers to extract contextual information from the mobile phone. The ContextPhone platform runs on off-the-shelf mobile phones using Symbian OS and Nokia series 60 smart phones [Raento et al., 2005].

2.7.1 ContextPhone Architecture

ContextPhone architecture comprises of four inter-connected components.

ContextPhone architecture is driven by four interconnected modules namely: Sensors, Communications, Customizable applications and System services. Figure 2.6 shows the architecture of the ContextPhone. A brief description of the four architectural modules follows. For further details refer to [Raento et al., 2005].

ContextPhone supports four types of logical sensors

Sensors: ContextPhone uses internal sensors to extract context of the mobile phone. This sensor data can then be stored, processed or transferred through the communication channel. The four kind of sensors that ContextPhone supports are:

- *Location*, including GSM cell identifier and GPS via Bluetooth GPS receiver.
- *User interaction*, including active applications, idel/active status, alarm profile, charger status and media capture.
- *Communication behavior*, including calls and call attempts, call recording and sent/received SMS.
- *Physical environment*, including surrounding Bluetooth devices, Bluetooth network availability and optical marker recognition.

ContextPhone supports local (IR and BT) and wide-area (GSM and GPS) communications

Communications: ContextPhone platform supports both local and wide-area communication. Infrared and Bluetooth is used for local communication while GSM and GPRS is used for wide-area communication. For gathering data, ContextPhone uploads data files automatically in the background using HTTP POST method. Any service

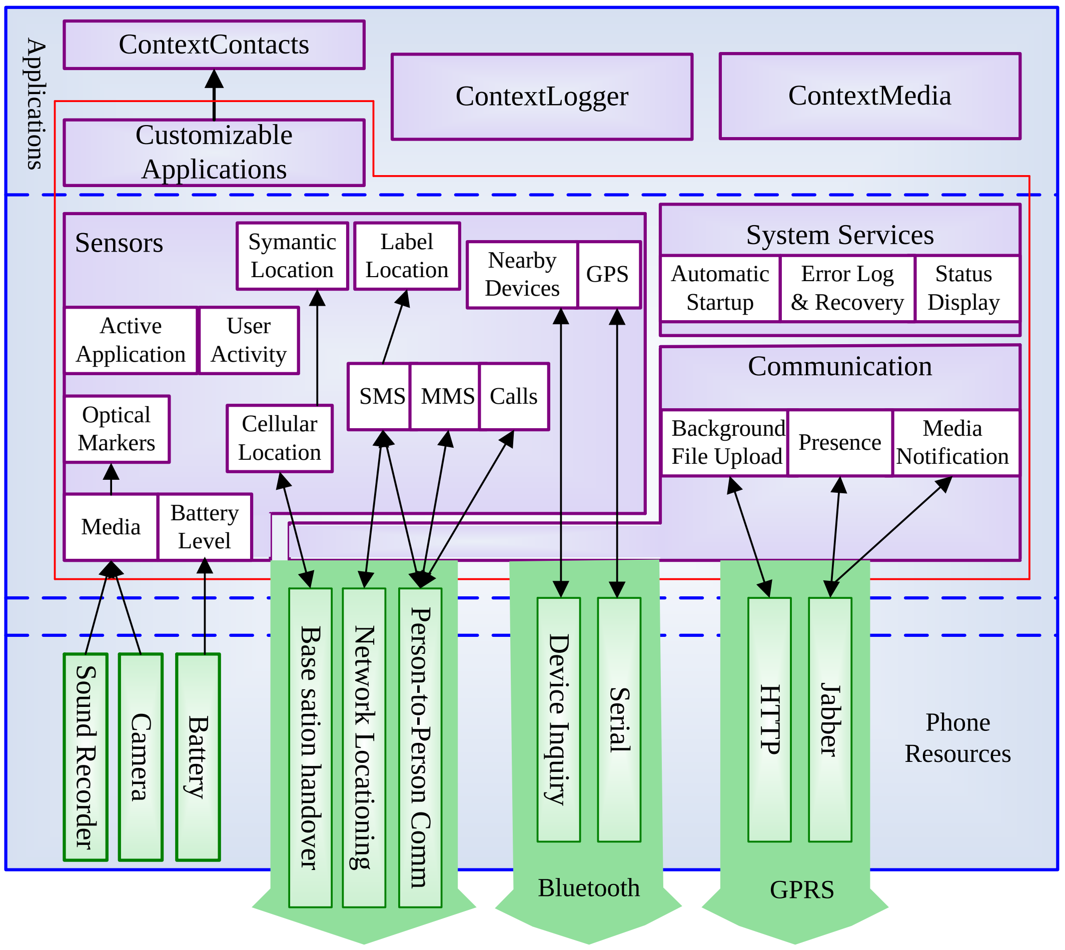


Figure 2.6: ContentPhone Architectural Diagram: This Figure has been reproduced from original in [Raento et al., 2005]

which uses messaging can be incorporated by developers since ContextPhone uses standard SMS/MMS messaging.

Customizable Applications: Customizable application have been developed using ContextPhone platform. Developers can use these applications which are typically customizable versions of built-in Series 60 **Contacts** and **Recent Calls** applications to add new features to person-to-person communication.

System Services: To support robustness, ContextPhone services add a feature that if a crash occurs, services and applications are restarted by a watchdog process. Context-

ContextPhone provides several system services

Phone components also support disconnected execution, queuing of operations, and storing of latest network information. The ContextPhone Architecture works on publish/subcribe model, in which components can publish or subscribe for context-events.

2.7.2 ContextPhone vs. iStuff Mobile

ContextPhone is targeted towards developers, however iStuff Mobile is targeted towards designers

The targeted users domains of ContextPhone and iStuff Mobile are clearly different. It is apparent that ContextPhone is targeted towards smart phone application developers, iStuff Mobile on the other hand targets smart phone interaction designers. Further, iStuff Mobile has a concrete mechanism to redirect the contextual information of the mobile phone to a ubiquitous computing environment. No such concrete infrastructure is provided by the ContextPhone platform.

iStuff Mobile is an interactive prototyping toolkit

iStuff Mobile provides an interactive prototyping environment to a designer in which the changes can be made to the mobile phone application behavior without the cumbersome compiling + download cycle. However, when using ContectPhone, changes in application behavior require re-compile and re-download of the mobile phone application.

ContextPhone focuses on using logical sensors e.g. location, user interaction etc. iStuff Mobile however enables designers to add additional physical sensors to gather contextual information and react to it.

Chapter 3

iStuff Mobile Architecture

"Each problem that I solved became a rule which served afterwards to solve other problems"

—Rene Descartes (1596-1650), "Discours de la Methode"

iStuff Mobile has been designed as compound architecture that allows designers to prototype interactions which may be beyond the hardware capabilities of a mobile phone

iStuff Mobile enables interaction designers to prototype interactions using mobile phones. These interactions may be beyond the hardware capabilities of available mobile phones. iStuff Mobile has been designed as a compound prototype architecture [Abowd et al., 2005] in which part of the software is distributed across separate computers. Figure 3.1 depicts the iStuff Mobile architecture. iStuff Mobile allows an interaction designer to augment mobile phone with external physical sensors (particle sensor module) that post sensor data onto the Event Heap 1.1.2—"Event Heap". This sensor data can be interpreted at the Patch Panel 1.1.6—"Patch Panel" level and high-level commands can be sent to the mobile phone over Bluetooth serial connection.

Direct communication b/w sensors and mobile phone would eliminate the advantages gained

One might argue that a direct communication channel between sensors and the mobile phone (e.g. through Bluetooth) would be more efficient. However, this would eliminate the possibility of dynamic configuration of the relationship between user activity and application feedback using the Patch Panel. When using iStuff Mobile, the delay

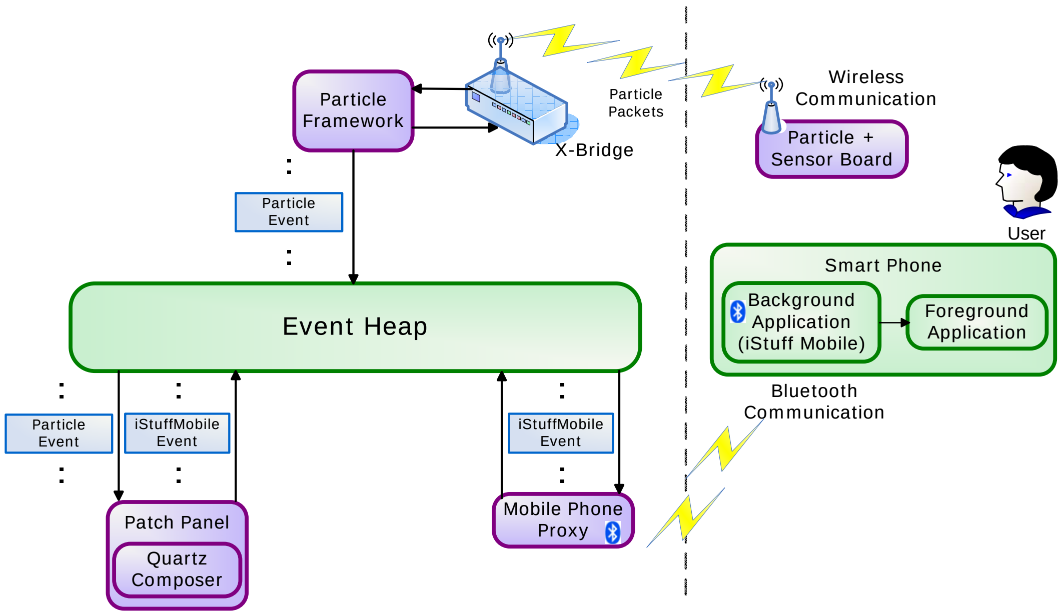


Figure 3.1: iStuff Mobile Architecture: An illustration of the iStuff Mobile architecture showing the Smart-Its particle framework proxy and the mobile phone proxy. The Event Heap supports publish/subscribe event communication. The Patch Panel intermediates between incompatible publishers and subscribers by translating events. This diagram shows only the Event Heap clients directly relevant to the iStuff Mobile architecture, but many other clients and proxies may be connected supporting distributed ubicomp interactions.

from Patch Panel

between the user action and application feedback is on the order of 10-29 ms which should not be a big performance concern.

Communication medium between components is Event Heap in iStuff Mobile architecture

The components in the iStuff Mobile architecture can be distributed across a room. These components use Event Heap infrastructure to communicate with each other. Since Smart-Its and mobile phone are not designed to communicate with the Event Heap, a proxy strategy is employed 1.1.3—“iStuff Components”.

As described in 1—“Introduction” iStuff Mobile extends the iStuff toolkit. This extension takes place in three important ways:

1. Support to use Smart-Its sensor network module is provided using the proxy strategy. This Proxy

(known as the *Particle Framework*) provides a GUI which allows configuration of sensors, filtering of sensor data and posting of sensor data onto the Event Heap.

2. A mobile phone application for Symbian Series 60 [Symbian Series 60] phones is provided. This application performs some system functions. It also allows designer to communicate with any foreground application running on the mobile phone.

3. A visual front-end is provided to rapidly specify Patch Panel mappings using Apple's Quartz Composer visual programming environment [Reiners, 2006].

iStuff Mobile is distributed under the Open Source Artistic License [OPI Artistic License]. The complete source code with documentation can be checked out through SVN from iStuff Mobile Developers Site[1] .

iStuff Mobile is open source and the complete code could be downloaded

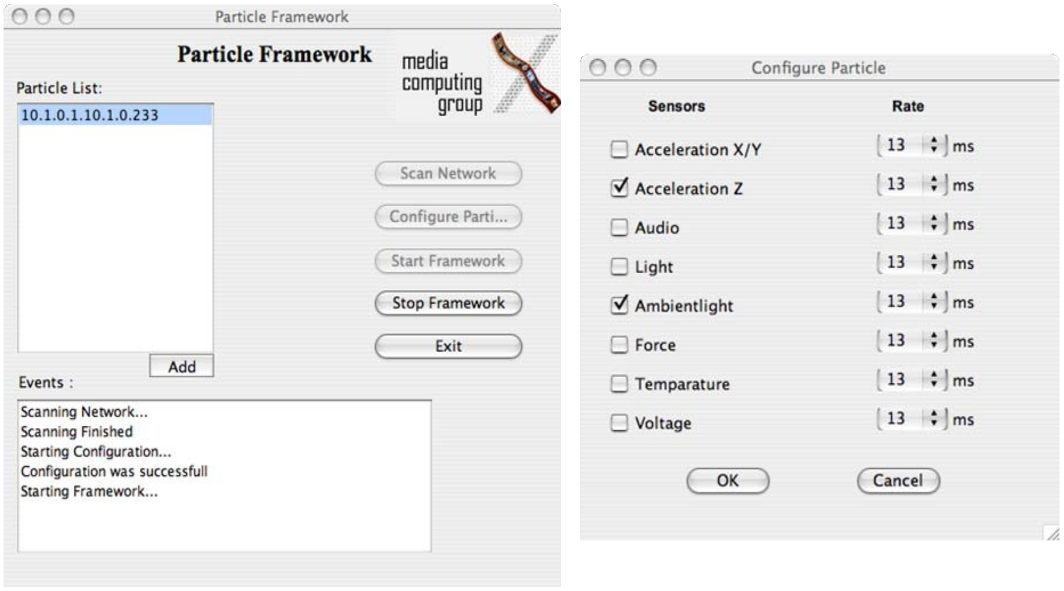
3.1 Particle Framework

Sensor network platforms have been proven to be valuable tools for rapid prototyping scenarios [Gellersen et al., 2004]. Typical prototyping scenarios using sensor enhanced interactions require the sensors to be small size, cheaper and widely available. For iStuff Mobile we decided to support Smart-Its (See 1.2—"Smart-Its Technology") sensor network module because of following features:

iStuff Mobile supports Smart-Its for prototyping sensor based interactions

- **Small Size:** A Smart-It is typically 45x27 mm in size.
- **Wireless Communication:** Smart-Its use wireless communication with up to 125kbit/sec bandwidth.
- **Easily Rechargeable:** Smart-Its use typical AAA batteries for power and are therefor easy to charge.

[1]http://developer.berlios.de/svn/?group_id=3259



Particle Framework

Particle List:
10.1.0.1.10.1.0.233

Events :

```
Scanning Network...
Scanning Finished
Starting Configuration...
Configuration was successfull
Starting Framework...
```

Sensors	Rate
☐ Acceleration X/Y	13 ms
☑ Acceleration Z	13 ms
☐ Audio	13 ms
☐ Light	13 ms
☑ Ambientlight	13 ms
☐ Force	13 ms
☐ Temparature	13 ms
☐ Voltage	13 ms

Figure 3.2: Left side of this figure shows the cross-platform GUI that allows for the discovery of and configuration of Smart-Its modules. By selecting "Start Framework" the user starts the proxy service that posts events to the Event Heap. Right side of this figure shows a subscreen of the interface on the left that allows users to configure the active sensors and their sampling rates.

- **Software Support:** Smart-Its come in with APIs for C/C++ and Java to establish communication between a computer and the sensor network module.
- **Ease of Use:** Smart-Its provide a RPC (Remote Procedural Call) interface to allow reconfiguration of sensors without modifying the code on the particle board.

The Particle Framework allows scanning of particles, reception and posting of sensor data onto the Event Heap

iStuffMobile makes use of these features to provide a cross-platform GUI known as the *Particle Framework* (See Figure 3.2) which enables scanning of particles in the network, configuration of sensors, reception of sensor data and posting of sensor data onto the Event Heap. Each Smart-It comes with an array of sensors (See Section 1.2.2—"Smart-Its Core and Sensor Boards"). Each of these sensors can be activated or deactivated over the air. Particle framework GUI allows a designer to enable/disable appropriate sensors to work with iStuff Mobile in a particular prototyping scenario. Implementation level details of particle frame-

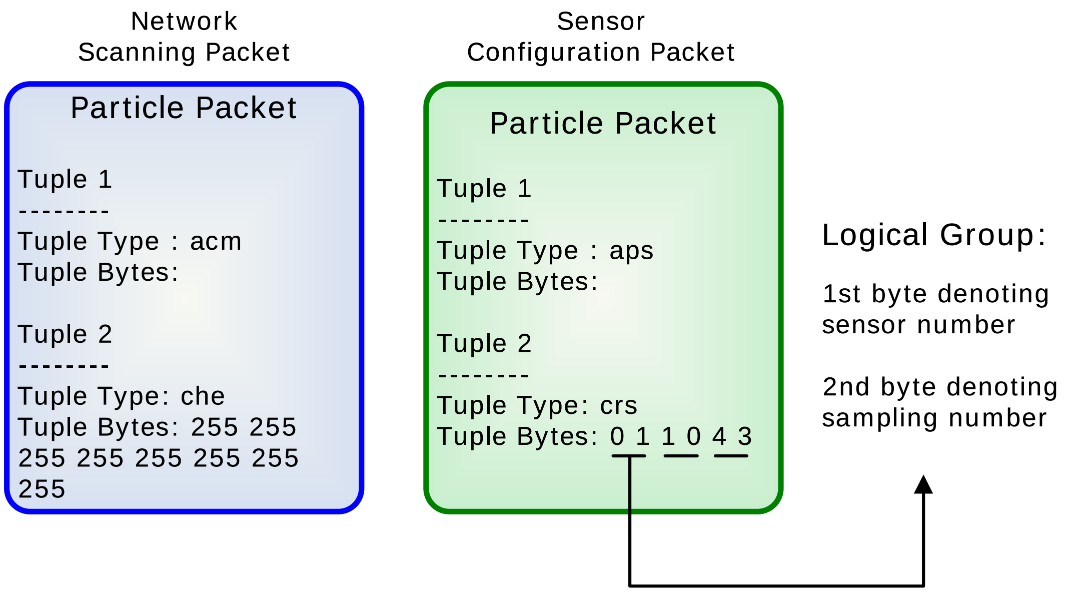


Figure 3.3: Left side of this figure shows a particle packet used for scanning network for particles. Right side of this figure shows a particle packet used for configuring sensor module on particle

work have been discussed in Section A—"Particle Framework Implementation".

iStuff Mobile architecture is flexible to allow incorporation of other sensor network platforms

iStuff Mobile has been designed in such a manner that Smart-Its sensor network platform can be replaced by any other sensor network platform. In order to incorporate a new sensor network platform into iStuff Mobile, a new Event Heap proxy has to be developed which would sent events on sensor module's behalf. Patch Panel mapping can then be altered to media communication between this new event type and the mobile phone proxy.

3.1.1 Scanning Network for Particles

Particle Framework scans network for particles by broadcasting a HELLO packet, which is responded by all particles

The particle framework scans the network for available particles and displays their source IDs. For scanning the network for particles, particle framework constructs a HELLO packet and broadcasts it to the network. All the particles inside the network respond to this packet by sending an acknowledgment to the particle framework. The particle

Sensor Number	Sensor Name
0	Acceleration X/Y axis
1	Acceleration Z axis
2	Audio
3	Light
4	Ambient Light
5	Force
6	Temperature
7	Voltage

Table 3.1: This table shows sensor numbers and the actual physical sensors they represent

framework then extracts the source Id of all the particles that responded to the HELLO packet and displays a list of available particles in the network. The format of HELLO packet is shown in Figure 3.3.

3.1.2 Configuring Particle Sensor Board

Definition:
Sampling Rate

SAMPLING RATE:
Sampling rate in particle context means the rate at which a Particle will transmit the data of a particular sensor

Particle Framework enables Smart-Its sensor board configuration

The particle framework also provides a GUI which enables a designer to activate/deactivate particular sensors on the particle sensor module (See right side of Figure 3.2). In order to do so, particle Framework constructs a configuration packet and transmits it to the particle selected for configuration. On reception of this configuration packet, the particle configures its sensor module accordingly and sends an acknowledgment to the particle framework.

Smart-Its sensor board configuration is achieved by transmitting a configuration packet to the Particle

Right side of the Figure 3.3 shows an example particle configuration packet. The first tuple of type ”aps” denotes that this packet is a configuration packet. The second tuple of type ”crs” contains the configuration data. In this case the sequence of bytes [01 10 43] means ”Configure sensor number 0 (Acceleration X/Y axis) with sampling number 1 (26

Sampling Number	Sampling Rate
0	13 ms
1	26 ms
2	52 ms
3	104 ms
4	208 ms
5	416 ms
6	832 ms
7	1664 ms
8	3328 ms
9	6656 ms
10	13312 ms
11	26624 ms
12	53248 ms
13	106496 ms
14	212992 ms
15	425984 ms

Table 3.2: This table shows sampling numbers and the actual sampling rates they represent

ms), configure sensor number 1 (Acceleration Z axis) with sampling number 0 (13 ms), and configure sensor number 4 (Ambient Light) with sampling number 3 (104 ms)". Rest of the sensors are not configured in this example and therefor they do not transmit any data. A list of sensor numbers and sampling numbers is given in Table 3.1 and Table 3.2 respectively.

3.1.3 Particle Packet to Particle Event

Particle Framework prevents redundant sensor data from being posted onto the Event Heap

As discussed in 1.2.3—"Smart-Its Communication", a Smart-It transfers the sensor data in form of packets (Particle Packets) to the X-Bridge. These packets are then received at the particle framework end over the LAN. A typical particle packet received from particle is shown on the left side of Figure 3.4. On receiving a particle packet, the particle framework compares the packet with the last packet received from the same source. If the two packets are different, the particle framework constructs an event, fills it with relevant data from particle packet, and posts it

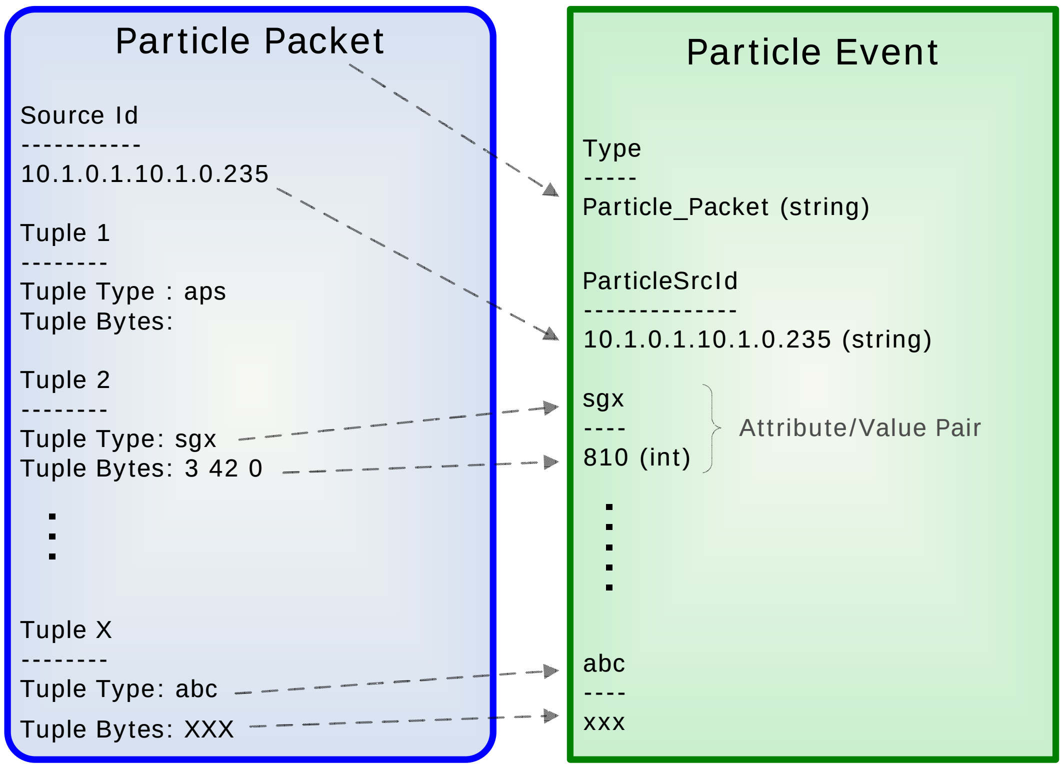


Figure 3.4: This figure shows transformation of particle packet to particle Event done by the particle framework (Only relevant information has been shown).

onto the Event Heap. A typical particle event is shown on the right side of Figure 3.4.

3.2 Mobile Phone Proxy

Mobile Phone talks to the Event Heap through a proxy in iStuff architecture

A normal mobile phone cannot talk to the Event Heap. In order to establish a communication between the Event Heap and the mobile phone, iStuff Mobile employs a proxy strategy. The proxy which communicates with the mobile phone and sends/receives events on its behalf is knows as the iStuff Mobile Proxy. Following are the three main functionalities of the mobile phone proxy:

1. Establish communication between a computer and an iStuff enabled mobile phone.

2. Relay events that are posted to the Event Heap by the Patch Panel to the mobile phone.

3. Listen to incoming actions from the mobile phone and react accordingly.

3.2.1 Mobile Phone Proxy and Mobile Phone Communication

iStuff Mobile Proxy connects to the mobile phone through bluetooth

Once the iStuff Mobile Proxy is launched, it connects to the Event Heap and waits for the mobile phone to connect to it. The mobile phone proxy establishes a connection with the mobile phone over bluetooth serial port.

After successfully establishing connection with the Event Heap and the mobile phone, the proxy listens to the Event Heap for relevant events and it also listens to the phone phone for relevant actions.

3.2.2 Event Relaying Process

The mobile phone proxy relays events from Event Heap to the mobile phone

Once communication has been established between the Event Heap and the mobile phone through iStuff Mobile Proxy, the proxy listens to the Event Heap for events of type ″iStuff Mobile″ with a mandatory field ″Command″. An ″iStuff Mobile″ type event with ″Command″ field can be divided into four categories:

- Events with only ″Command″ field. In this case, the mobile phone proxy simply extracts the value of the ″Command″ field and sends it over the bluetooth serial port to the mobile phone.

- Events with ″Command″ and ″Path″ fields. In this case, the proxy extracts the value of ″Command″ and ″Path″ fields. Then it sends the value of the ″Command″ field to the mobile phone, followed by the length of the value of the ″Path″ field, followed by the value of the ″Path″ field.

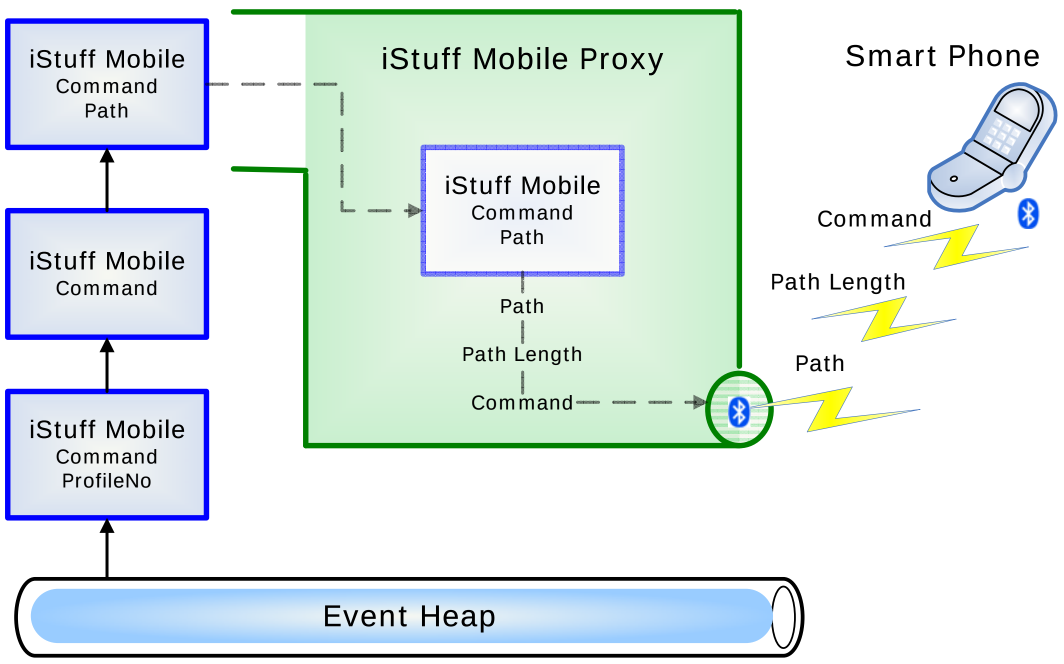


Figure 3.5: The figure shows an example of relaying event from Event Heap to the iStuff Mobile enabled smart phone

- Events with "Command", "Repeat", "ScanCode" and "Code" fields, which represent a key press event. In this case, the proxy extracts these fields and send them to the mobile phone in the order they are stated here. This sequence cause the mobile phone to simulate a key press on the foreground application.

- Events with "Command" and "ProfileNo" fields. In this case, the proxy extracts the value of "Command" and "ProfileNo" fields. Then it sends the value of the "Command" field to the mobile phone, followed by the value of the "ProfileNo" field.

Figure 3.5 shows an example or event relaying process.

3.2.3 User Action Relaying Process

Mobile phone relays user activity (Key

The communication between the Event Heap and the mobile phone is not one sided. Hence, after establishing the

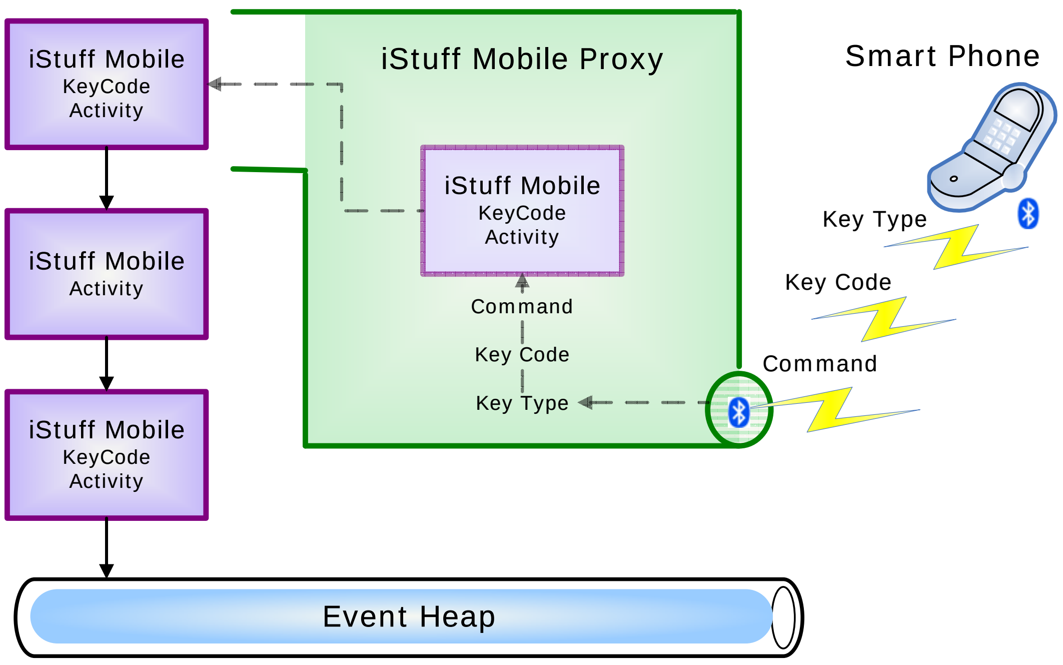


Figure 3.6: The figure shows an example of relaying user activity (Key Presses) from the iStuff Mobile enabled smart phone to the Event Heap

communication between the Event Heap and the mobile phone, the iStuff Mobile Proxy also listens for any user activity that the mobile phone wants to forward to the Event Heap. For the current version of iStuff Mobile, the user activity forwarded by the mobile phone is User interaction with the GUI in form of Key Press Events.

Presses) to the
Event Heap

Key Type Value	Key Activity
1	KeyPress
2	KeyUp
3	KeyDown

Table 3.3: This table shows key type values and the corresponding user activity they represent

The mobile phone proxy listens to the bluetooth serial port for any incoming commands. If the command for key press is received, the proxy expects the ascii value of the key, as well as well as the type value of key to be followed. On receiving the expected values, the proxy posts an ”iStuff Mobile” type event containing ”KeyCode” (ascii value) and

"Activity" (interpretation of type value) fields on to the Event Heap. Table 3.3 shows a list of type values and the corresponding key activities and Figure 3.6 shows an example of relaying key presses from the mobile phone to the Event Heap.

3.3 iStuff Mobile Smart Phone Applications

The iStuff Mobile architecture creates a division between the mobile phone *background* application and the mobile phone *foreground* application.

Definition: *Foreground Application*

FOREGROUND APPLICATION:
The foreground application is designed to be used by the user, the person that is testing the prototype design during user evaluation.

Definition: *Background Application*

BACKGROUND APPLICATION:
The background application is designed to simplify the work of interaction designer, the person that creates functional prototypes. The background application is not directly visible to the prototype user.

3.3.1 Background Application

iStuff Mobile background application is responsible for communicating with the proxy, OS of the mobile phone and the foreground application of the mobile phone

The background application is a part of iStuff Mobile that is provided for the designers by the framework. Interaction designers can remotely executes commands on the mobile phone by sending commands in form of iStuffMobile events to the iStuff Mobile Proxy. The proxy relays these commands to the background application via a Bluetooth connection (See Section 3.2.2—"Event Relaying Process"). The background application relays these commands to the foreground application or the operating system as appropriate. The background application is also capable of intercepting user activity on the foreground application, such as

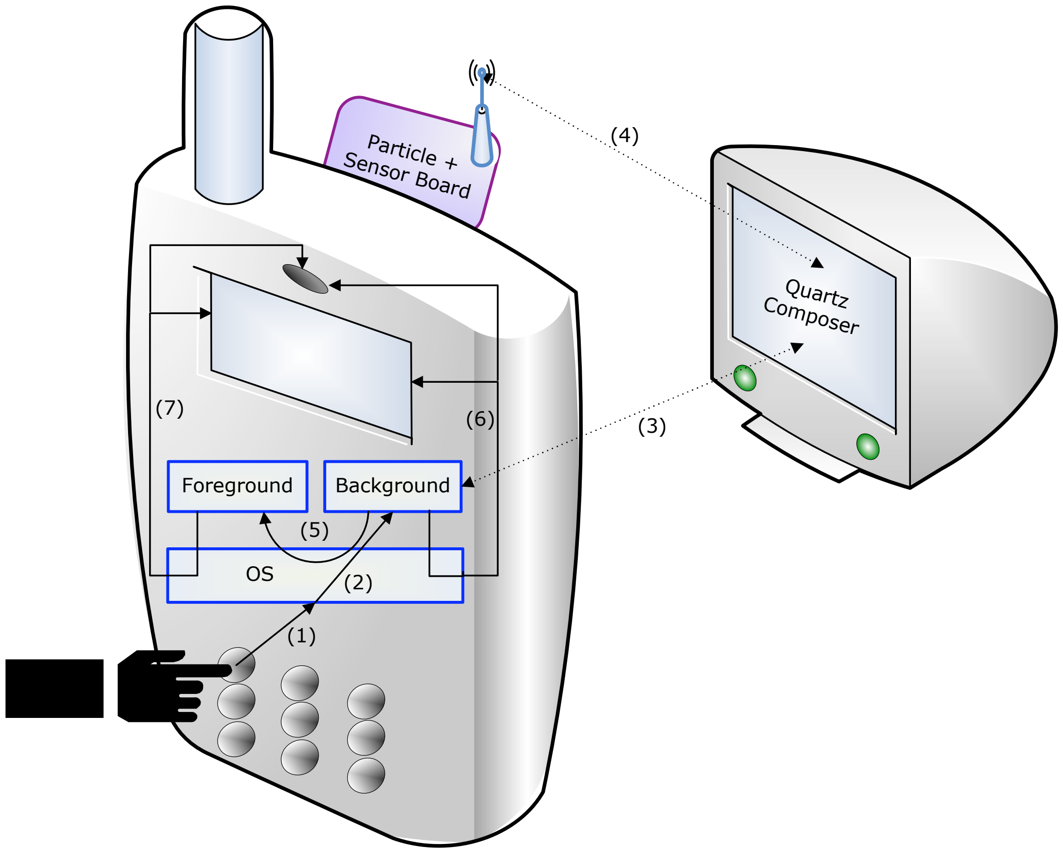


Figure 3.7: An illustration of the key functional components in the iStuff Mobile architecture. (1) The key press from the user is sent to the mobile phone operating system. (2) Typically, the operating system (OS) would send key presses to the foreground application, however the background application has registered to intercept the event notification from the foreground. (3) The background application notifies Quartz Composer of user activity through a Bluetooth connection. (4) Meanwhile, the Smart-Its sensor board is also reporting sensor data. Quartz Composer interprets the data and issues a high-level command to the background such as a (5) key press to the foreground or (6) producing system output such as sound feedback. (7) The foreground application responds to key press by updating the UI or producing other system output.

key presses, which are relayed to the proxy over the bluetooth and subsequently posted as events onto the Event Heap (See Section 3.2.3—"User Action Relaying Process"). Figure 3.7 portrays an example user interaction with the mobile phone.

This architecture enables designers to prototype interaciStuff Mobile can be used to prototype interactions with new or readily available mobile phone applications

tions with existing applications (such as the built-in Address Book and Calender application), or the new ones created using Symbian C++, Python, or Macromedia Flash Lite. The iStuff Mobile background application was designed to include following features but is not limited to them. New features may be added later:

1. **Bluetooth Communication:** communicate with the iStuff Mobile proxy through a low-latency wireless communication channel.
2. **Sound Playback:** trigger available sound files to be played or stopped.
3. **Vibrator Control:** trigger the vibrator to start and stop.
4. **Key Capture Capability:** intercept key events from the foreground application and relay them to the proxy for processing.
5. **Foreground Application Key Simulation:** relay key events to the foreground application.
6. **Launch External Application:** launch any application among the available application on the mobile phone.
7. **Close External Application:** close any running application on the mobile phone.
8. **Profile Control:** programmatically change the ring profile of the mobile phone.
9. **Backlight Control:** turn the backlight ON and OFF programmatically.
10. **Run Application in Background:** send the current foreground application to the background.
11. **Camera Control:** use the camera on the mobile phone for taking pictures, videos, interactions using motion estimation such as the Sweep technique [Ballagas et al., 2005], and recognizing visual markers such as Visual Codes [Rohs and Gfeller, 2004].

	Bluetooth Communication	Sound Playback	Vibrator Control	Key Capture Capability	Foreground Application Key Simulation	Launch External Application	Close External Application	Profile Control	Backlight Control	Run Application in Background	Camera Control
Symbian Series 60	○✓	○✓	○	○✓	○✓	○✓	○✓	✓ a	○✓ b	○✓	○✓ c
Java (J2ME)	○	○		○ d		○ e					○
Windows Mobile 5.0 SmartPhone	○	○	○	○	○	○	○		○	○	○

○ Available ✓ Implemented

Table 3.4: A preliminary analysis of API support for different smart phone platforms. Our prototype background application was written for Symbian Series 60, but it could be ported to Windows Mobile 5.0 Smartphone. This analysis was accumulated from a survey of system documentation and developer forums.

[a] Profile Change implemented in the iStuff Mobile application using Application Launch and Foreground Application Key Simulation.
[b] Partially implemented (only turning ON functionality).
[c] Sweep and Point & Shoot interactions implemented in a separate Series 60 application, still need to integrate into iStuff Mobile background application.
[d] Available only when the Java application is the foreground application.
[e] Available only for Java applications inside the same package.

Current background application prototype has been developed for Symbian platform, but Windows Mobile 5.0 platform is a good candidate

First background application have been developed using Symbian Series 60 SDK. However, iStuff Mobile is not limited to Symbian Series 60 phones. Table 3.4 shows a preliminary analysis of necessary API support required to port iStuff Mobile background application on alternative smart phone platforms. From the table one can clearly deduce that Windows Mobile 5.0 SmartPhone would be a great candidate to port iStuff Mobile background application since it shares many of same capabilities as Symbian Series 60. Porting iStuff Mobile background application to Java 2 Micro Edition (J2ME) platform would not be useful, since it would be lacking critical functionalities. Theoretically, iStuff Mobile background application can also be ported to Linux-based mobile phone platforms (e.g. Motorola E680i), but currently no public SDKs are provided for development on these phone. Open source development efforts on these phones are still in their early stages.

3.3.2 Foreground Application

Communication between foreground and background application takes place through key events

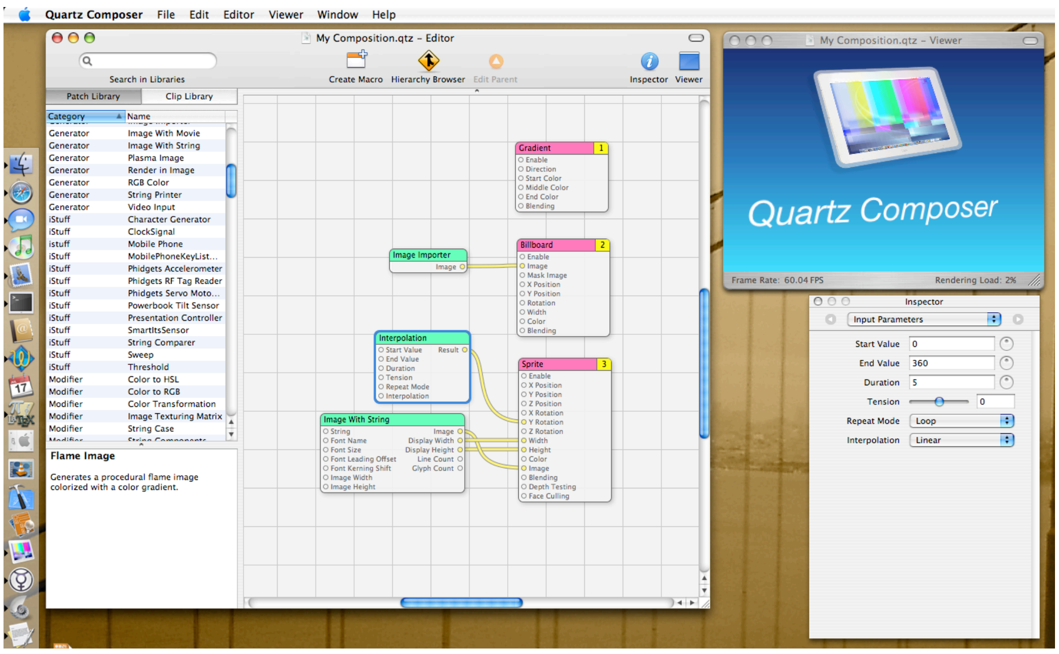
The foreground application in iStuff Mobile architecture is the application that user sees and interacts with on the mobile phone. iStuff Mobile background application has been designed to interact with any foreground application. The communication between the background application and the foreground application takes place primarily through system events (key press simulation).Designers are expected to prototype their own mobile phone application using rapid prototyping solutions such as Flash Lite [Macromedia Flash Lite], or a scripting language like Python [Python for Series 60]. Alternatively, designers can use Java [Java Platform, Micro Edition] or native code to program their own application with which iStuff Mobile background application would interact. Lastly, designers can use existing mobile phone applications, that come with the mobile phone (such as Address Book and Calander) to prototype custom interactions, despite the fact that these applications were not originally designed to accommodate new interaction.

Designer can built their own mobile phone application which can communicate with the iStuff Mobile background application

3.4 Visual Programming Support

Apple's Quartz Composer has been adapted to define Patch Panel mappings for iStuff components

Quartz Composer [Apple Quartz Composer] is a visual programming environment that is part of the Xcode development environment that comes with Mac OS X 10.4 "Tiger" (see Figure 3.8). It uses a cable patching metaphor to establish data and control flow between different components, establishing a composition. The editor is live, and changes made in the workspace are immediately functional without any compilation steps. In iStuff Mobile, work of Rene Reiners [Reiners, 2006] extends the Quartz Composer environment to define Patch Panel mapping using Visual Programming. Library components for all iStuff proxies have been added to this extension of Quartz Composer. New processing modules have also been included in this extension which are particularly useful for physical prototyping scenarios.



Category	Name
Generator	Image With Movie
Generator	Image With String
Generator	Plasma Image
Generator	Render in Image
Generator	RGB Color
Generator	String Printer
Generator	Video Input
iStuff	Character Generator
iStuff	ClockSignal
istuff	Mobile Phone
iStuff	MobilePhoneKeyList...
iStuff	Phidgets Accelerometer
iStuff	Phidgets RF Tag Reader
iStuff	Phidgets Servo Moto...
iStuff	Powerbook Tilt Sensor
iStuff	Presentation Controller
iStuff	SmartItsSensor
iStuff	String Comparer
iStuff	Sweep
iStuff	Threshold
Modifier	Color to HSL
Modifier	Color to RGB
Modifier	Color Transformation
Modifier	Image Texturing Matrix
Modifier	String Case

Flame Image

Generates a procedural flame image colorized with a color gradient.

Input Parameters

Start Value	0
End Value	360
Duration	5
Tension	0
Repeat Mode	Loop
Interpolation	Linear

Figure 3.8: Apple's Quartz Composer is a visual programming environment designed to support rapid creation of 3D interactive visualizations. iStuff Mobile extends it to provided prototyping solution for physical interfaces. The main window provides a library of components that the designer can drag onto the workspace and connect with other components. The preview window in the top right of the image is a live 3D animation of the composition and immediately shows the impact of the designer's changes. The inspector window in the bottom right allows the designer to adjust parameters and settings of the different library components used in the composition.

Chapter 4

Prototyping with iStuff Mobile

"Interesting - I use a Mac to help me design the next Cray."

—Seymoure Cray (1925-1996) when he was told that Apple Inc. had recently bought a Cray supercomputer to help them design the next Mac

iStuff Mobile is a powerful prototyping toolkit which enables interaction designers to prototype interaction using mobile phones. To present the usability and domain coverage of this toolkit, this chapter shows the interaction that have been prototyped using the iStuff Mobile toolkit.

4.1 Recreating inspiring mobile phone interaction

Famous mobile phone interactions have been recreated

This section talks about recreating interactions that have been proposed in previous literature. To demonstrate the utility of iStuff Mobile, three of the classical inspiring mobile phone interactions have been recreated:

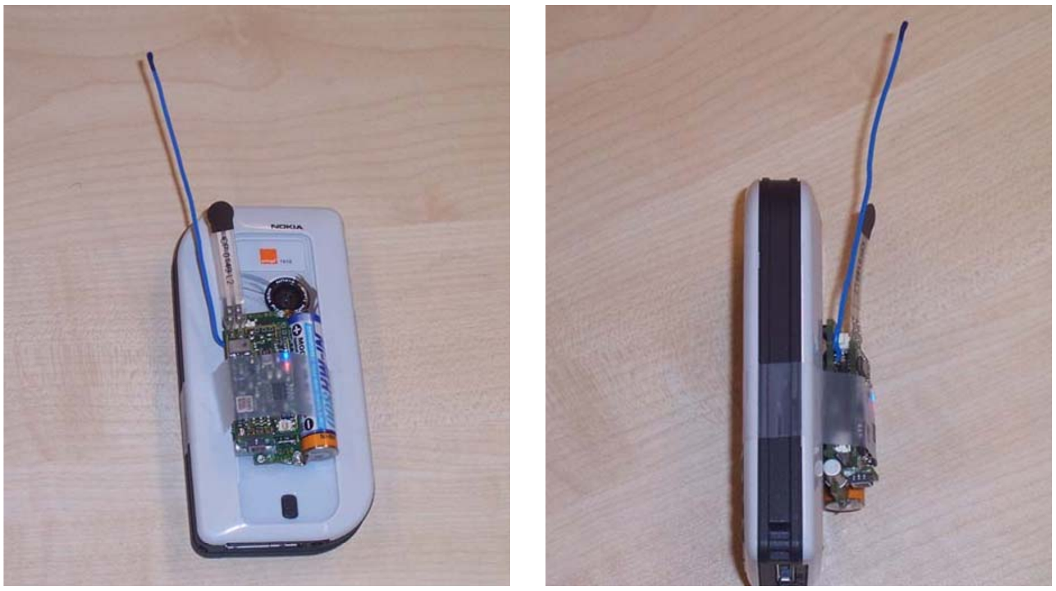


Figure 4.1: This figure shows a nokia series 60 mobile phone augmented with a Smart-It(Particle) on the backside. A designer can attach a Particle to the mobile phone in whatever position suitable. The Smart-Its board in pictures contains a 3D accelerometer, light sensor, temperature sensor, audio sensor, pressure sensor and a voltage sensor.

4.1.1 Tilt Scrolling

Tilt Scrolling technique

[Harrison et al., 1998] introduced a tilt scrolling technique for mobile devices. The idea was to scroll through any type of sequential list using the tilt. The implementation of this technique included use of a Palm Pilot equipped with pressure and tilt sensors. To activate the tilt scrolling, user had to press the sides of the handheld device with his/her thumb and forefinger. The degree of tilt was used to control the speed of scrolling i.e. the more the user tilted the device, the faster it scrolled. Using iStuff Mobile, this example interaction has been recreated.

Tilt Scrolling has been recreated using iStuff Mobile

Recreating this interaction involved taping a Particle to the back of the mobile phone (See Figure 4.1) and configuring it to transmit data of Force and Acceleration(Z-axis) sensors. The interpretation of sensor data was done at the Patch Panel level using the Quartz Composer GUI and appropriate signals were sent to the iStuff Mobile Proxy. Figure 4.2

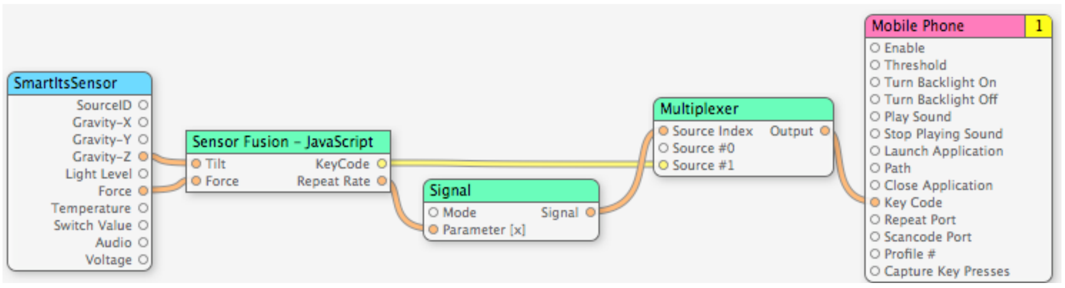


Figure 4.2: The Quartz Composer implementation of the tilt-scrolling interaction from [Harrison et al., 1998]. JavaScript nodes in Quartz Composer can manipulate data using simple scripts. In this patch, it verifies that the force sensor is pressed and detects various thresholds in the Z-direction of the gravity sensor. ″KeyCode″ represents the appropriate key (up or down arrow) depending on the current tilt. ″Repeat Rate″ specifies how often the Signal node should fire. The Signal node output (0 to 1 at regular intervals) specifies the Source Index for the multiplexer. Source #0 (which defaults to 0) represents no key pressed, and Source #1 represents the key specified from the JavaScript node.

which is the screen shot of the patch used for tilt scrolling, explains how all the pieces fit together.

4.1.2 Changing Ringing Profile

Profile Change according to user context

[Schmidt et al., 1999] introduced a technique in which the mobile phone detects the context of a user and changes the ringing profile accordingly. For example, when the mobile phone is in the hand of a user, the ringing profile is changed automatically to vibrator only, since an audio notification is unnecessary. The context information was gathered using the TEA boards (See Section 2.3—“The TEA project”). Currently, ringing profiles exists on all Symbian series 60 phones. Following profiles were defined by the original authors and recreated on the phone for the experiment:

- **Hand:** If the phone is in hand of the user, audio indication is not needed. Hence, phone rings by vibrating.
- **Table:** It is assumed that the user is in a meeting situation. Incoming calls and messages are indicated with a gentle sound.

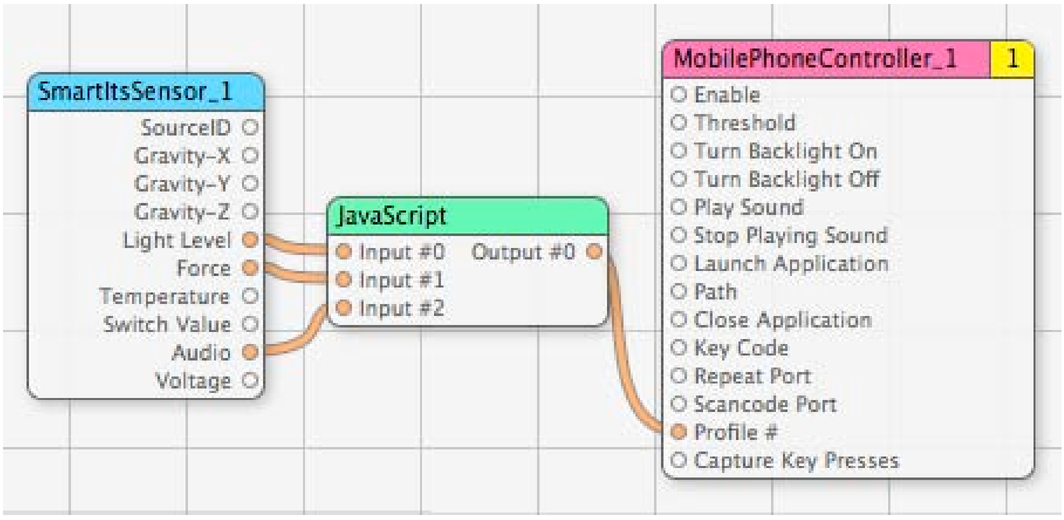


Figure 4.3: The Quartz Composer implementation of the context based profile change described in [Schmidt et al., 1999]. The "JavaScript" node changes the profile based on the values of Light, Audio and Force sensors

- **Silent:** In this case it is assumed that the phone has been put away in a box or a briefcase, therefore it should remain silent.
- **Outside:** It is assumed that the user is in a crowd. Therefore the ringing volume goes as high as possible and the vibrator alarm is also turned on.
- **General:** Where none of the above cases apply.

The profile change technique has been recreated using iStuff Mobile

Using iStuff Mobile, handedness is detected using the pressure sensor on the Smart-Its. Difference between Table and Silent profile is detected using the light sensor, since inside a briefcase it would be dark. Finally Outside scenario is detected using the audio sensor on the Smart-Its. Figure 4.3 shows the patch panel composition of this prototyping scenario.

4.1.3 Tilt Typing

Original TiltText interaction uses tilt to type characters on the mobile phone

[Wigdor and Balakrishnan, 2003] introduces a technique using which, text entry on the mobile phone has been simplified. This technique works by typing a letter from the

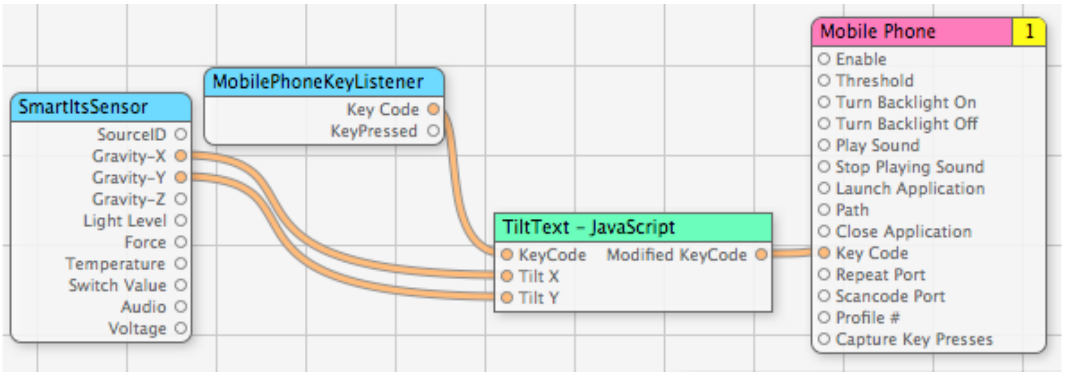


Figure 4.4: The Quartz Composer implementation of the TiltText [Wigdor and Balakrishnan, 2003] first registers to incept keys on the mobile phone. Then it transforms the numeric keys to the corresponding alphabet characters based on the tilt thresholds (detected using acceleration sensors). The other keys on the mobile phone are allowed directly to be sent back to the mobile phone.

numeric keypad using a combination of key presses and device tilt. If the phone is tilted left and a key is pressed, the first letter is typed, tilting upwards activates the second letter, tilting to the right activates the third letter, tilting downwards activates the fourth letter (Applicable for keys '7' and '9') and no tilt activates the standard numeric key character. This technique has been proven to be faster than MultiTap and comparable to the T9 dictionary based technique. Using iStuff Mobile, the prototype of TiltText was recreated easily. Figure 4.4 shows the patch panel composition for TiltText scenario.

TiltText has been recreated using iStuff Mobile

4.2 Ubiquitous Computing Prototyping Scenarios

iStuff Mobile is not limited to creating interactions that are local to the mobile phone. This section will demonstrate how iStuff Mobile simplifies prototyping ubiquitous computing scenarios.

iStuff Mobile simplifies Ubicomp prototyping

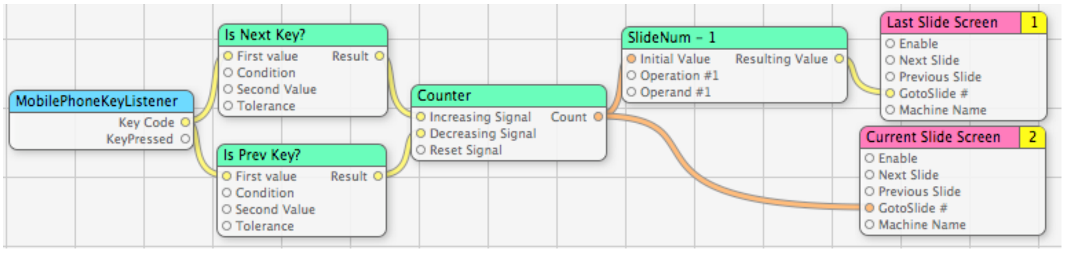


Figure 4.5: This Figure shows the Quartz Composer implementation of multi-screen presentation patch. The far left is the ”MobilePhoneKeyListener” module receives the key presses from the iStuff Mobile Proxy. On the far right are the two instances of the same PowerPoint presentation, each running on a separate machine.

4.2.1 Multi-Screen Presentation Control

iStuff Mobile has been used to create a multi-screen presentation interaction with three different screens showing current, previous and next slide respectively

Mobile phones have recently emerged as popular presentation controller. Salling Clicker [Salling Clicker] converts a mobile phone or PDA into a two-way wireless remote control for a single computer. But in ubicomp environments (e.g. interactive workspaces [Johanson et al., 2002a]) the presence of multiple large displays can be taken advantage of to enhance a presentation. Using iStuff Mobile a multi-screen presentation interaction has been created. In this scenario one screen displays the current slide (slide that is being presented) and a second screen displays the slide history (previous slide). By pressing a key on the mobile phone the user can move the presentation in forward or backward direction. In this scenario, the user is remotely controlling two PowerPoint [Microsoft PowerPoint] application. These applications are controlled by two different proxies running on separate machines in interactive workspace. Each proxy listens to an event containing the machine name of the machine that its running on, so that each machine can be individually controlled. Figure 4.5 shows the patch panel composition of multi-screen presentation scenario.

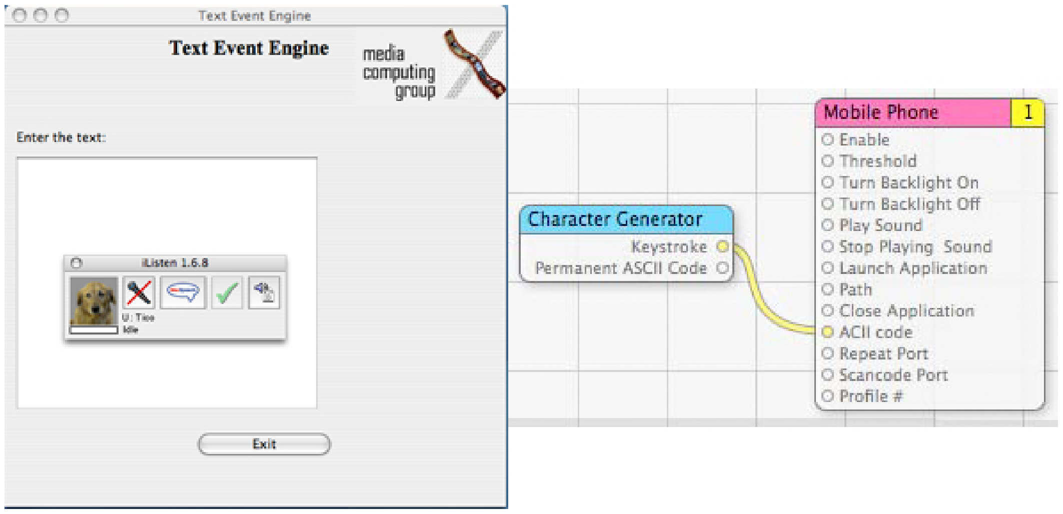


Figure 4.6: The left side of this figure shows a Java program that generates events for each character that is typed into it. The floating window upon this program is the iListen software for continuous speech recognition on Mac OS X. The right side shows the patch panel composition in which the events generated by Java ″Text Event Engine″ are recognized by the ″Charactor Generator″ module and transfered to the mobile phone. This patch panel composition can be used to type a message using computer keyboard onto the mobile phone or even dictate a message to be typed on the mobile phone.

4.2.2 Keyboard Redirection

Projects like PointRight [Johanson et al., 2002b] and ARIS [Biehl and Bailey, 2004] allow users to redirect mouse and keyboard input to different machines in a room. Using iStuff Mobile a prototype interaction has been created which demonstrates redirection of keyboard input to the mobile phone. Figure 4.6 shows the patch panel composition of this prototype.

4.2.3 Speech Text

Using iStuff Mobile, keyboard input can be redirected to the mobile phone

Using the keyboard input redirection, a prototype has been created in which user can dictate text to a mobile phone using continuous speech recognition. This is done by pointing the iListen software for continuous speech recognition to type inside the text area of ″Text Event Engine″ (See Fig-

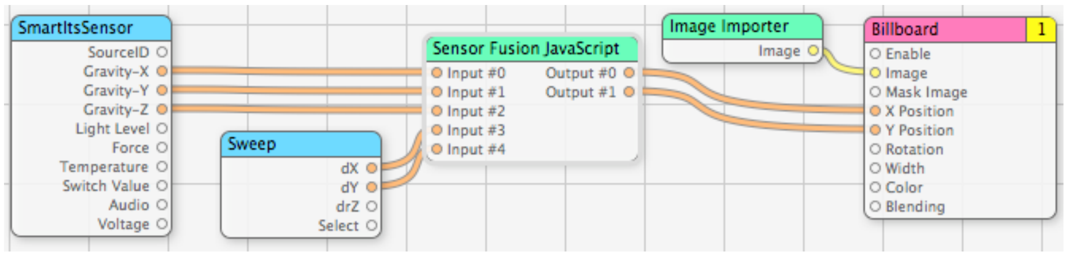


Figure 4.7: The Quartz Composer implementation for combining accelorometer data with camera-based motion detection to improve the motion detection accuracy. The ”Sensor Fusion JavaScript” node implements the algorithm to combine the sensor values in a meaningful way. The JavaScript logic can be modified at runtime to test and refine the sensor fusion strategy. The standard ”Billboard” node of Quartz Composer displays an image to the screen (e.g a cursor arrow). The output of the sensor fusion algorithm in the JavaScript node controls the position of the billboard on the screen.

ure 4.6). The processing power of phone phone these days is not sufficient for continuous speech recognition but using iStuff Mobile a functional prototype has been created.

4.2.4 Phone As Accurate Pointing Device

iStuff Mobile allows mobile phone to be used as an accurate pointing device for large public displays

The Sweep technique [Ballagas et al., 2005] allows a mobile phone to be used as a relative pointing device for large public displays. It uses camera-based estimation algorithm for motion detection. Since mobile phones are low-powered devices, the motion estimation algorithm is not perfect and suffers from estimation errors. Using iStuff Mobile, the motion information from the mobile phone camera can been combined with accelerometer data from the Smart-Its sensors to improve the motion estimation, as show in Figure 4.7. This allows mobile phone to serve as an accurate pointing device, for example when interacting with large public displays.

Chapter 5

Summary and future work

"Ask her to wait a moment - I am almost done."

—Carl Friedrich Gauss (1777-1855), while working, when informed that his wife is dying

5.1 Summary

iStuff Mobile is a rapidly prototyping toolkit which enables interaction designer to explore novel interactions involving mobile phone. iStuff Mobile extends the iStuff Framework to provide following four architectural components:

iStuff Mobile provides support for Smart-Its

- Support for physical sensors. iStuff Mobile provides a Java proxy which communicates the Smart-Its, gathers the sensor data and posts it onto the Event Heap. A typical Smart-It contains 8 physical sensors including light, acceleration (3D), sound, voltage, temperature and force sensors.

iStuff Mobile provides a mobile phone application

- A mobile phone application for Symbian Series 60 platform. This application communicates with the Event Heap and executes high-level commands on the mobile phone.

Mobile phone communicates through a proxy

- A proxy program which communicates the the mobile phone using Bluetooth Serial port and receives/sends events on its behalf.
- Extensions for Apple's Quartz Composer to enable patch panel mapping through visual programming.

It has been demonstrated that using iStuff Mobile designers can create low fidelity functional prototypes in no time. iStuff Mobile not only allows designers to explore new mobile phone interaction in ubicomp environments but it also simplifies development of sensor-enhanced interaction.

5.2 Future work

iStuff Mobile is still an active research topic. Following are the enhancements that would be

- Built new features into the mobile phone background application (e.g. Camera Control, Turning Backlight OFF).
- The camera based interaction techniques used in 4.2.4—"Phone As Accurate Pointing Device" have been implemented as a separate Symbian Series 60 application. One task would be to integrate these techniques into the iStuff Mobile smartphone application.
- iStuff Mobile smartphone background application is only available for Symbian Series 60 platform at the moment. We plan to port this application to other platforms like Windows Mobile 5.0 and Linux based mobile phone platforms.
- Provide a mechanism to initiate inter-process communication with the smartphone background application allowing the foreground application to post events. This could be done using publish/subscribe mechanism in Symbian.

By making this tool available through open source to the Ubicomp research community, we hope to advance the pace of innovation and improve the quality of interface designs in ubiquitous computing.

iStuff Mobile is an Open Source project

Appendix A

Particle Framework Implementation

"Your development cycle is much faster because Java is interpreted. The compile-link-load-test-crash-debug cycle is obsolete."

—James Gosling

Particle Framework is a GUI that allows a designer to scan for Smart-Its in the network, configure sensors according to the need and post the sensor data onto the Event Heap (See Section 3.1—"Particle Framework"). Particle Framework has been developed keeping the cross-platform property in focus. Therefore, the platform used is Java (J2SE) [Java Platform, Standard Edition]. A Jave API is provided by Teco which allows communication with the Smart-Its. This Java API can be downloaded from Teco Particle Website[1] .

Particle Framework has been developed using Java (J2SE)

Figure A.1 shows the static structure of Particle Framework. Following is a detail discussion of these classes:

[1] http://particle.teco.edu/software/index.html

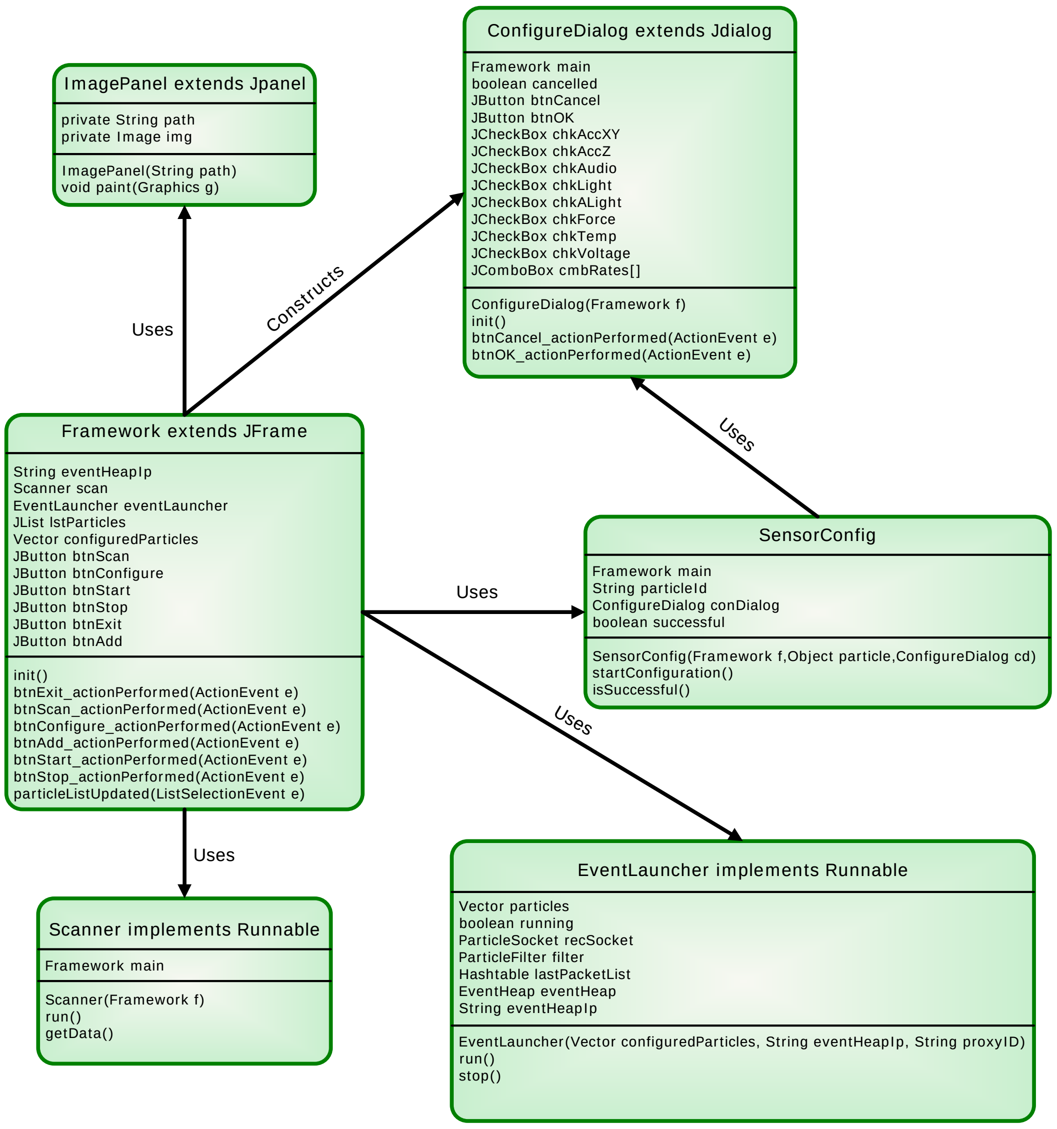


Figure A.1: Class Diagram of Particle Framework. Only the relevant methods and member variables have been displayed.

A.1 Framework class

`Framework` class constructs the main GUI of the Particle Framework

The `Framework` class in the Particle Framework is the main class that instantiates objects of all the other classes according to need (See Figure A.1). The `Framework` class is also responsible for constructing the main GUI of the Particle Framework by extending the `JFrame` class provided by J2SE (See Figure 3.2). The main functionality of the `Framework` class can be portrayed as follows:

- Construct the main graphical user interface and display it to the user.
- When the user clicks the ″Scan Network″ button, construct an instance of the `Scanner` class and execute it in a separate thread.
- When the user clicks on any Particle Id displayed in the `lstParticles` list, enable the ″Configure Particle″ button on the GUI.
- When the user clicks on the ″Configure Particle″ button, create an instance of `ConfigureDialog` class displaying the configuration options to the user.
- When the `ConfigureDialog` is closed, check if the user cancelled/accepted the configuration. This is done by checking a boolean variable `cancelled` inside the `ConfigureDialog` class (Details in Section A.3—“ConfigureDialog class”).
- If the `ConfigureDialog` was closed successfully, construct an instance of `SensorConfig` class and pass the Id of the selected Particle in the `lstParticles` list and the instance of `ConfigureDialog` class as constructor parameters.
- Enable the ″Start Framework″ button if `isSuccessful()` method of `SensorConfig` class returns `true`.
- When the user clicks ″Start Framework″ button, construct an instance of `EventLauncher` class in a separate thread and pass list of configured Particles `configuredParticles` and the event heap IP address `eventHeapIp` (received as command-line argument) as constructor parameters, disable the ″Scan Network″ button, disable the ″Configure Particle″ button, disable the ″Start Framework″ button and enable the ″Stop Framework″ button.
- When the user clicks ″Stop Framework″ button, call `stop()` method on instance of `EventLauncher`

```
ParticleSocket socket = new ParticleSocket(5556); //5556
denotes outgoing socket
ParticlePacket hello = new ParticlePacket();  //construct a
new Particle packet
 short data[] = {255, 255, 255, 255, 255, 255, 255,
255};  //set up the destination address as a broadcast
address

short empt[] = {};
hello.aclAdd("acm", empt, 0);
hello.aclAdd("che", data, 0);
socket.send(hello);
socket.close();
```

Figure A.2: Code for broadcasting HELLO packet into the network in order to scan the Particles. The `ParticleSocket` and `ParticlePacket` classes are the part of Java API provided by Teco. Two tuples are added to the `hello` packet before `socket.send(hello)` is called. The ″acm″ tuple with no data denotes that this packet is a HELLO packet and ″che″ tuple contains the broadcast address.

class, enable all the disabled button and disable the ″Stop Framework″ button.

A.2 Scanner class

`Scanner` class is responsible for scanning network for Particles

The `Scanner` class in Particle Framework is responsible for scanning the network for Particles (See Section 3.1.1—“Scanning Network for Particles”). This class implements the Java `Runnable` interface which allows it to run in a separate thread.

`Scanner` class broadcasts a HELLO packet into the network which is responded by all Particles

As soon as an object of this class is instantiated, the `run()` method of this class is invoked. Inside the `run()` method a HELLO packet is broadcasted into the network and it is expected that all the Particles will respond to this packet. Code snip A.2 shows how a HELLO packet is constructed and broadcasted into the network.

Once the HELLO packet is broadcasted into the network,

```
ParticleSocket socket = new ParticleSocket(5555);  //open a
socker for receiving packets from "Particles"

socket.setAutoAck(0);  //dont send an acknowledgment
ParticlePacket pck = null;
socket.setBlocking(0); //dont block the thread
pck = socket.receive(socket);  //receive a packet
if (pck != null)
{
    ParticleSrcId src = pck.getSrcId();  //get the Source Id
of the particle
    if (!main.listmodelParticles.contains(src.toString()))
   {

main.listmodelParticles.addElement(src.toString());  //if the
id doesnot exists in the list already, add it to the list
   }
}
```

Figure A.3: Code for handling Particle response to the HELLO packet (Only relevant part has been shown). The `getData()` method opens a socket for reception, checks an incoming packet, extracts the Particle Id from the packet and finally adds the Particle Id to the list displayed on the `Framework` GUI. The Particle Id is added to the list only if it does'nt already exists.

the particles start to respond to it. In order to check which particles have responded, `getData()` method of `Scanner` class is called at the end of `run()` method. Code snip A.3 shows how the response of HELLO packet from Particles is handled.

Ids of the Particles responding to HELLO packet are collected

A.3 ConfigureDialog class

The `ConfigDialog` class is a secondary GUI which pops up when the user clicks on the "Configure Particle" button on the main GUI (See right side of Figure 3.2). The purpose of this GUI is to provide a user with options to configure sensor module on a particular Particle. Once the user has selected a particular configuration, he can either click

`ConfigureDialog` provides a GUI to configure sensors on a particular Particle

```
ParticlePacket packet = new ParticlePacket(); //create a new
packet to be sent to the "Particle"
Vector sensorData = new Vector();      //holds the sensor
numbers and sampling rates selected by the user
short data[] = new short[0];
packet.aclAdd( "aps", data, 0 ); //add an empty "aps" tuple to
the packet

if(conDialog.chkAccXY.isSelected() == true) //if X/Y axis
acceleration sensor is selected
{
    sensorData.add(new Short("0")); //add the sensor no. to the
vector. Sensor no. 0 is X/Y axis acceleration sensor
    sensorData.add(new
Short((short)conDialog.cmbRates[0].getSelectedIndex())); // add
the index of selected transfer rate to the vector
}                            -
                             -
                             -
                             -
                             -
data = new short[sensorData.size()];
for(int i=0;i<sensorData.size();i++)
 data[i] = ((Short) sensorData.get(i)).shortValue(); //collect
the selected sensor and rates

packet.aclAdd( "crs", data, 0 ); //add the configuration data
to the packet as a tuple
```

Figure A.4: Code showing construction of a configuration packet. This code first adds and empty "aps" tuple to the `packet`. Then it checks which sensors were selected for configuration and adds their sensor numbers and sampling numbers to the `sensorData` Vector (only check for Acceleration sensors has been shown, rest are similar). Finally the data from the `sensorData` Vector is transferred into the `data` array and added to the `packet` as an "crs" tuple

on "Cancel" button in which case the `cancelled` boolean variable becomes true and the GUI disappears from the screen, or he can click the "OK" button in which case the `cancelled` boolean variable remains false and the GUI disappears from the screen.

```
ParticleSrcId src = new
ParticleSrcId(particleId.toString());
ParticleSocket sndSocket = new ParticleSocket(5556);
ParticleSocket recSocket = new ParticleSocket(5555);

recSocket.setBlocking(0);
sndSocket.sendAcked(recSocket, packet, src); // if this
returns without exception the particle is configured
successful = true;

sndSocket.close();
recSocket.close();
```

Figure A.5: Code for transmitting configuration packet to a Particle. This code starts by opening two sockets. `sndSocket` is for sending the configuration packet `packet` and `recSocket` is for receiving acknowledgment from the Particle. Finally the `packet` is sent by calling `sndSocket.sendAcked(recSocket, packet, src)`. If this function call returns without an exception, the Particle is configured.

A.4 SensorConfig class

After the sensor module configuration dialog disappears, the `Framework` class checks if `cancelled` boolean member of `ConfigureDialog` is `true` or `false`. If `cancelled` is `false` an instance of `SensorConfig` class is created passing the selected particle Id, the instance of `ConfigureDialog` and a reference to `Framework` class.

`SensorConfig` constructs a configuration packet and transmits it to the Particle

Once the instance of `SensorConfig` class is created, its `startConfiguration()` method is invoked. Inside this method, the selected sensor numbers and sampling numbers used to construct a configuration packet which is to be transmitted to the particle. Code snip A.4 shows how the configuration packet is constructed. For more information on sensor numbers and sampling numbers see Section 3.1.2—"Configuring Particle Sensor Board".

As soon as the Configuration packet is constructed, it is sent to the Particle who's id was passed as argument to the

`SensorConfig` class constructor, requiring an acknowledgment from the Particle. Code snip A.5 shows how a configuration packet is sent to the Particle.

A.5 EventLauncher class

`EventLauncher` class receives sensor data, filters it and posts it onto the Event Heap

The `EventLauncher` class in responsible for receiving packets from all the configured Particles inside the network, filtering packets for redundancy and posting the sensor data onto the Event Heap. An instance of `EventLauncher` class is constructed when "Start Framework" button on the main GUI is clicked and there are one or more configured Particles inside the `configuredParticles` vector in class `Framework`. The `EventLauncher` class implements the java `Runnable` interface which allows it to run in a separate thread.

As soon as the `EventLauncher` object is constructed, its `run()` method is invoked. Inside the `run()` method, the first piece of code starts to receive packets from all the configured Particles. Code snip A.6 shows how packets are received from all the Particles that have been configured through Particle Framework.

Redundant sensor values are discarded

After a single packet is received inside the `run()` method, the code checks if the packet transmitted by the Particle contains the same values as the previous one transmitted by the same Particle. If the values are same the packet is simply discarded. Code snip A.7 shows how the redundant packet detection algorithm runs.

If the packet received from a Particle is different from the previous one received from the same Particle, the sensor data is extracted from the packet, it is interpreted and each sensor value is converted to a single integer value, and finally the converted sensor values are posted onto the Event Heap.

```
int filterMode = P_FILTER.FILTER_CONCAT_AND |
P_FILTER.FILTER_TYPE_ID;
filter = new ParticleFilter("Id Filter");
ParticleSrcId recFrom = new ParticleSrcId((String)
particles.get(0));
filter.add(filterMode,recFrom.toFilter()); //receive
packets from this particle

if(particles.size() > 1) //if the particle list contains
more than one particle
{
     filterMode = P_FILTER.FILTER_CONCAT_OR |
                        P_FILTER.FILTER_TYPE_ID;

     for (int i = 1; i < particles.size(); i++)
     {
        recFrom = new ParticleSrcId((String)
particles.get(i));
        filter.add(filterMode, recFrom.toFilter()); //add
all the particles to the reception list
     }
 }

recSocket = new ParticleSocket(5555); //open a socket for
reception of packets from particles

currPacket = recSocket.receiveFiltered(recSocket,filter);
```

Figure A.6: Code for receiving packets from configured Particles. The code starts by setting a `filter` which is an instance of `ParticleFilter` class to receive packets from all the Particles that are in `particles` vector. Then it opens `recSocket` socket for receiving packets. Finally the call `recSocket.receiveFiltered(recSocket,filter)` receives a single packet based on the filter

A.6 ImagePanel class

Class `ImagePanel` extends the java `JPanel` to provide a panel in which an image is loaded. An instance of this class is used by the `Framework` class to load the i10 department

`ImagePanel` class draws a panel with an image inside

```
if(currPacket != null)
{
   currId = currPacket.getSrcId(); //get the Source Id of the packet received
   ParticlePacket lastPacket = (ParticlePacket)
lastPacketList.get(currId.toString()); //get the previous packet transmitted by
this Source from the hashtable

   if(lastPacket != null) //if the previous packet transfered from this particle
exists
   {
      ParticleTuple currPacketTuple = currPacket.firstAcl(); //get the 1st tuple
from the current packet
      ParticleTuple lastPacketTuple =
lastPacket.findFirstAcl(currPacketTuple.getAclType()); //get the 1st tuple from
the previous packet

      while (currPacketTuple != null)
      {
         short[] currPacketTupleData = currPacketTuple.toArray(); //get data of
the current packet tuple
         short[] lastPacketTupleData = lastPacketTuple.toArray(); //get data of
the previous packet tuple

         for(int i=0;i<currPacketTupleData.length;i++)
         {
            if(currPacketTupleData[i] != lastPacketTupleData[i])   //if the data
is not equal
            {
                 same = false;
                 break;
            }
          }
          if(!same)
            break;

          currPacketTuple = currPacket.nextAcl(currPacketTuple); //get the next
tuple from the current packet
          lastPacketTuple = lastPacket.nextAcl(lastPacketTuple); //get the next
tuple from the previous packet
        }
     }
     else
       same = false;
```

Figure A.7: Code for detecting redundant packets. This code starts by retrieving previous packet transmitted by the same Particle from the `lastPacketList` hashtable. The code then compares the current packet and the previous packet tuple by tuple and byte by byte inside a tuple. If difference is found at any byte, the `same` boolean variable is set to false and comparison stops.

logo onto the GUI. The constructor of this class simply takes the path of an image and constructs a Panel containing that image.

Appendix B

iStuff Mobile Proxy Implementation

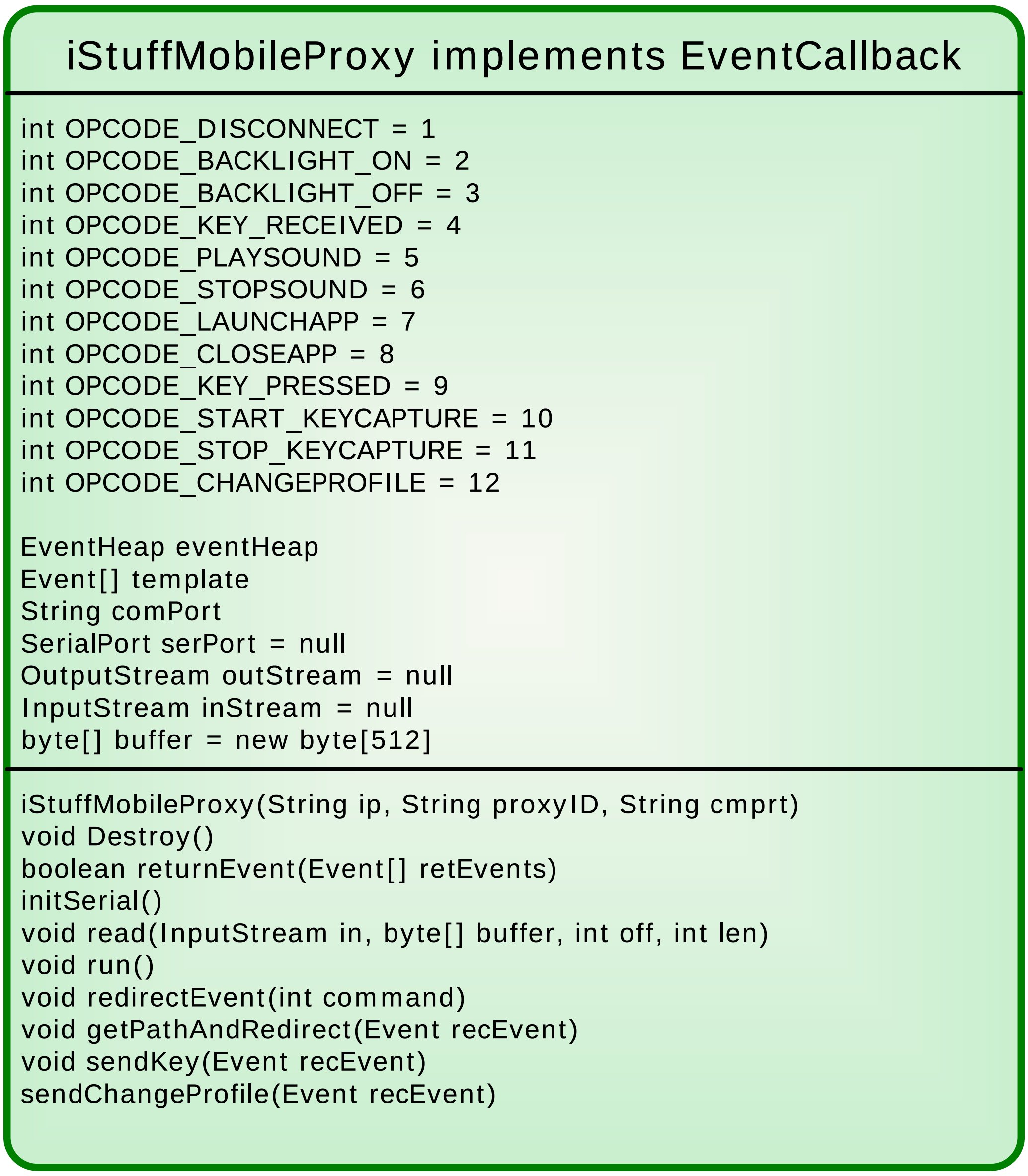
"That's what's cool about working with computers. They don't argue, they remember everything and they don't drink all your beer."

—Paul Leary

iStuff Mobile Proxy is a command line program which receives/posts events on the behalf of the mobile phone. Since mobile phone is not designed to communicate directly with the Event Heap, iStuff Mobile Proxy implements the interface which talks to the mobile phone through Bluetooth serial port and establishes communication between the mobile phone and the Event Heap. iStuff Mobile Proxy has been developed in Java (J2SE) [Java Platform, Standard Edition] to make it available on different platforms. iStuff Mobile Proxy comprises a single class `iStuffMobileProxy` who's static structure is portrayed in Figure B.1. Following are the main functions of the `iStuffMobileProxy` class:

iStuff Mobile Proxy has been implemented using Java (J2SE)

- Initiate a serial port communication with the mobile phone and register to receive events of type "iStuff-Mobile" from the Event Heap.



```
iStuffMobileProxy implements EventCallback

int OPCODE_DISCONNECT = 1
int OPCODE_BACKLIGHT_ON = 2
int OPCODE_BACKLIGHT_OFF = 3
int OPCODE_KEY_RECEIVED = 4
int OPCODE_PLAYSOUND = 5
int OPCODE_STOPSOUND = 6
int OPCODE_LAUNCHAPP = 7
int OPCODE_CLOSEAPP = 8
int OPCODE_KEY_PRESSED = 9
int OPCODE_START_KEYCAPTURE = 10
int OPCODE_STOP_KEYCAPTURE = 11
int OPCODE_CHANGEPROFILE = 12

EventHeap eventHeap
Event[] template
String comPort
SerialPort serPort = null
OutputStream outStream = null
InputStream inStream = null
byte[] buffer = new byte[512]

iStuffMobileProxy(String ip, String proxyID, String cmprt)
void Destroy()
boolean returnEvent(Event[] retEvents)
initSerial()
void read(InputStream in, byte[] buffer, int off, int len)
void run()
void redirectEvent(int command)
void getPathAndRedirect(Event recEvent)
void sendKey(Event recEvent)
sendChangeProfile(Event recEvent)
```

Figure B.1: Class Diagram of iStuff Mobile Proxy. Only the relevant methods and member variables have been displayed.

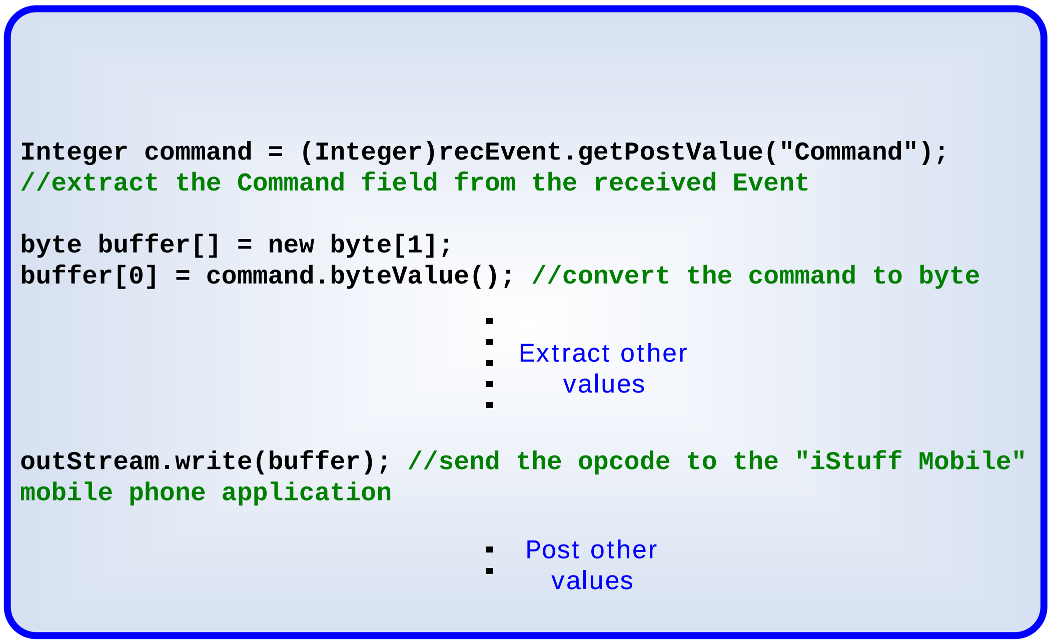
- Decode the events received from the Event Heap and redirect the decoded form of events to the mobile phone.
- Listen to incoming key presses from the mobile phone, encapsulate them into an event and post them onto the Event Heap.

```
eventHeap = new EventHeap(ip);
template = new Event[1];
template[0] = new Event("iStuffMobile"); //the Events to be
fetched should be of type iStuffMobile

template[0].addField("Command", Integer.class,
FieldValueTypes.FORMAL, FieldValueTypes.FORMAL); //the events
to be fetched should have a field Command of an Integer value
type
eventHeap.registerForEvents(template,this); //register to
receive events of type template
                           -
                           -
                           -
                           -
                           -

CommPortIdentifier portId =
CommPortIdentifier.getPortIdentifier(comPort); //get the port
ID for the port name received through command line
if (portId == null)
{
  throw new NullPointerException("no com port identifier");
}
serPort = (SerialPort)portId.open("iStuffMobile", 5000); //open
the serial Port
outStream = serPort.getOutputStream(); //get output stream to
the serial port
inStream = serPort.getInputStream(); //get input stram to the
serial port
```

Figure B.2: This code first shows how a template event `template` is used to register for events of type "iStuffMobile" containing an `Integer` type field "Command". The method call `eventHeap.registerForEvents(template,this)` tells the Event Heap that `boolean returnEvent(Event[] retEvents)` method of `iStuffMobileProxy` should be called whenever an event of type `template` is posted onto the Event Heap. Second part of the code shows fetching the COM port identifier using COM port name `comPort` which was passed as command-line argument, opening the serial port and finally getting the I/O streams to the port.

```
read(inStream, buffer, 0, 1); //read the opcode received by the
"iStuff Mobile" mobile phone application

switch (buffer[0]) {
case OPCODE_KEY_PRESSED:
read(inStream, buffer, 0, 4); //if the opcode is
OPCODE_KEY_PRESSED, 4 bytes will follow. 2 bytes denoting the
keycode and 2 bytes denoting the keytype
Event keyEvent = new Event("iStuffMobile");

char keyCode = 0;
keyCode |= buffer[0];//fill the keycode from two bytes into a
char
keyCode <<= 8;
keyCode |= buffer[1];

char type = 0;
type |= buffer[2]; //fill the keytype from two bytes into a char
type <<= 8;
type |= buffer[3];

switch(type){

       case 1: // type 1 denotes key was pressed
         keyEvent.setPostValue("Activity", "KeyPress");
             keyEvent.setPostValue("KeyCode", new
Integer(keyCode));      break;
       case 2: // type 2 denotes key was released
         keyEvent.setPostValue("Activity", "KeyUp");
         break;
       case 3: // type 3 denoted key was hit
         keyEvent.setPostValue("Activity", "KeyDown");
         break;
        default:
         System.out.println("Unrecognized Key Type");}

if (eventHeap.isConnected())
   eventHeap.putEvent(keyEvent); //post the received key press
from the "iStuff Mobile" mobile
```

Figure B.3: This code first listens to a single byte on the incoming stream. If the byte received is `OPCODE_KEY_PRESSED`, it expects 4 more bytes to follow containing key code and key type respectively. Finally the 2 bytes of key code and key type are filled into 2 chars `keyCode` and `type`, the `type` is interpreted according to Table 3.3, and both `keyCode` and `type` are encapsulated in an event and posted onto the Event Heap



```
Integer command = (Integer)recEvent.getPostValue("Command");
//extract the Command field from the received Event

byte buffer[] = new byte[1];
buffer[0] = command.byteValue(); //convert the command to byte
                                       .
                                       .  Extract other
                                       .      values
                                       .

outStream.write(buffer); //send the opcode to the "iStuff Mobile"
mobile phone application

                                       .  Post other
                                       .    values
```

Figure B.4: Code showing extraction of ”Command” field from an event and redirecting its value to the mobile phone through the output stream `outStream` (Extraction of any other field would be similar)

B.1 iStuffMobileProxy class

`iStuffMobileProxy` class registers with the Event Heap to receive events of type ”iStuffMobile”

The first function of `iStuffMobileProxy` when started, is to register to the Event Heap for receiving events of type ”iStuffMobile”. This is done by defining a template event and telling the Event Heap to deliver events matching the template to the proxy. The next function of the `iStuffMobileProxy` is to initialize a serial port communication with the COM port which was passed as a command-line argument to the program. Code snip B.2 shows how the proxy registers for events and how the communication streams are initialized.

`iStuffMobileProxy` class listens for incoming key events from mobile phone and relays them to the Event Heap

Once the `iStuffMobileProxy` is initialized, its `run()` method is called from the `main()`. The `run()` method starts to listen to the incoming stream for the mobile phone key press events. The key events are relayed by the mobile phone in form of 5 bytes. The first byte represent the opcode sent by the mobile phone, next 2 bytes represent the key code of the key pressed and the last 2 bytes represent the key type. Code snip B.3 shows how key code and key type are received from the mobile phone, embedded into an event and posted onto the Event Heap.

Opcode Representation	Opcode Value
OPCODE_DISCONNECT	1
OPCODE_BACKLIGHT_ON	2
OPCODE_BACKLIGHT_OFF	3
OPCODE_KEY_RECEIVED	4
OPCODE_PLAYSOUND	5
OPCODE_STOPSOUND	6
OPCODE_LAUNCHAPP	7
OPCODE_CLOSEAPP	8
OPCODE_KEY_PRESSED	9
OPCODE_START_KEYCAPTURE	10
OPCODE_STOP_KEYCAPTURE	11
OPCODE_CHANGEPROFILE	12

Table B.1: This table shows the values a ″Command″ field can contain in an event received by the iStuff Mobile Proxy

All the events of type ″iStuffMobile″ are sent to the `iStuffMobileProxy` through a call-back method

Since the `iStuffMobileProxy` class has registered for events matching a template, whenever such an event occurs a callback method `boolean returnEvent(Event[] retEvents)` is called with the event as a parameter. The event can be of one of the types described in Section 3.2.2—"Event Relaying Process″. After receiving such an event, the `iStuffMobileProxy` tries to decode the ″Command″ field inside it. The ″Command″ field can have one of the value shown in Table B.1. Code snip B.4 shows how a field can be extracted from an event. Incase a key event is received from the Event Heap (See Section 3.2.2—"Event Relaying Process″), the integer values of ″Code″, ″Repeat″ and ″ScanCode″ are converted to six byte values and then transmitted to the mobile phone, which is done as shown in code snip B.5.

```
byte buffer1[] = new byte[6];
//coversion of Code, Repeat and ScanCode fields from integer to 2
bytes

buffer1[0] = 0;
buffer1[0] |= (0xFF00 & repeat) >> 8;
buffer1[1] = 0;
buffer1[1] |= (0x00FF & repeat);

buffer1[2] = 0;
buffer1[2] |= (0xFF00 & scancode) >> 8;
buffer1[3] = 0;
buffer1[4] |= (0x00FF & scancode);

buffer1[4] = 0;
buffer1[4] |= (0xFF00 & code) >> 8;
buffer1[5] = 0;
buffer1[5] |= (0x00FF & code);
```

Figure B.5: Code showing conversion of `code`, `scancode` and `repeat` variables from three integers to six bytes.

Appendix C

iStuff Mobile SmartPhone Application Implementation

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg."

—Bjarne Stroustrup

iStuff Mobile smartphone application is actually the background application running on the smartphone. A communication channel is established between the background application and the Event Heap through iStuff Mobile proxy. As explained earlier in Section 3.3.1—"Background Application", the first prototype of background application has been implemented for Symbian Series 60 platform [Symbian Series 60]. The SDK that has been used to implement the iStuff Mobile smartphone background application is the Nokia Series 60 SDK [Nokia Series 60]. This section explain how the functionalities described in Section 3.3.1—"Background Application" have been implemented on the mobile phone. Figure C.1 shows the static structure of the iStuff Mobile smartphone background application. Following is a detail discussion on implementation of classes relevant to the functionality of the background application:

iStuff Mobile Smartphone application has been implemented using Symbian series 60 SDK from Nokia

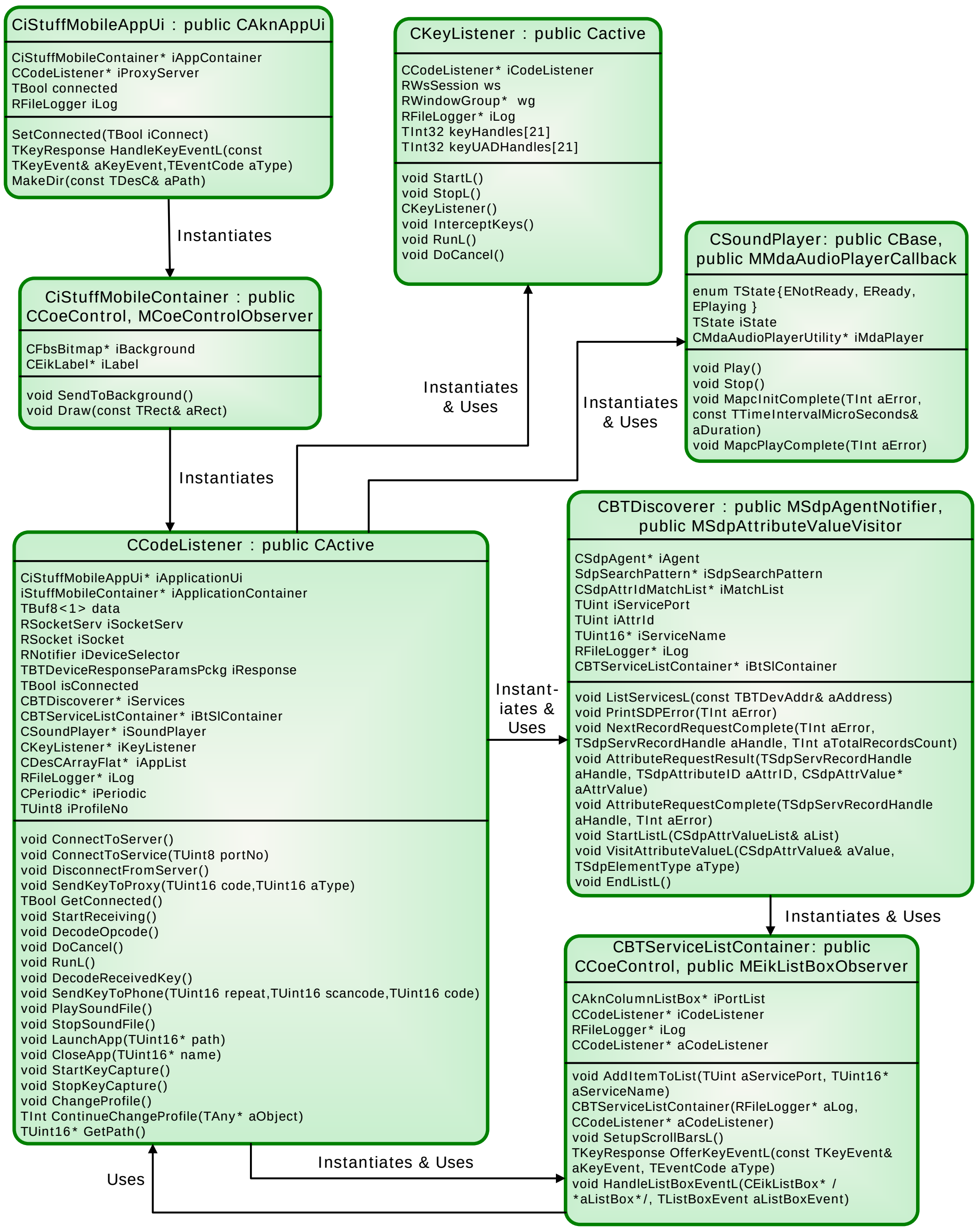

CiStuffMobileAppUi : public CAknAppUi
CiStuffMobileContainer* iAppContainer
CCodeListener* iProxyServer
TBool connected
RFileLogger iLog
SetConnected(TBool iConnect)
TKeyResponse HandleKeyEventL(const TKeyEvent& aKeyEvent,TEventCode aType)
MakeDir(const TDesC& aPath)
Instantiates
CKeyListener : public Cactive
CCodeListener* iCodeListener
RWsSession ws
RWindowGroup* wg
RFileLogger* iLog
TInt32 keyHandles[21]
TInt32 keyUADHandles[21]
void StartL()
void StopL()
CKeyListener()
void InterceptKeys()
void RunL()
void DoCancel()
CSoundPlayer: public CBase, public MMdaAudioPlayerCallback
enum TState{ENotReady, EReady, EPlaying }
TState iState
CMdaAudioPlayerUtility* iMdaPlayer
void Play()
void Stop()
void MapcInitComplete(TInt aError, const TTimeIntervalMicroSeconds& aDuration)
void MapcPlayComplete(TInt aError)
CiStuffMobileContainer : public CCoeControl, MCoeControlObserver
CFbsBitmap* iBackground
CEikLabel* iLabel
void SendToBackground()
void Draw(const TRect& aRect)
Instantiates & Uses
Instantiates & Uses
Instantiates
CCodeListener : public CActive
CiStuffMobileAppUi* iApplicationUi
iStuffMobileContainer* iApplicationContainer
TBuf8<1> data
RSocketServ iSocketServ
RSocket iSocket
RNotifier iDeviceSelector
TBTDeviceResponseParamsPckg iResponse
TBool isConnected
CBTDiscoverer* iServices
CBTServiceListContainer* iBtSlContainer
CSoundPlayer* iSoundPlayer
CKeyListener* iKeyListener
CDesCArrayFlat* iAppList
RFileLogger* iLog
CPeriodic* iPeriodic
TUint8 iProfileNo
void ConnectToServer()
void ConnectToService(TUint8 portNo)
void DisconnectFromServer()
void SendKeyToProxy(TUint16 code,TUint16 aType)
TBool GetConnected()
void StartReceiving()
void DecodeOpcode()
void DoCancel()
void RunL()
void DecodeReceivedKey()
void SendKeyToPhone(TUint16 repeat,TUint16 scancode,TUint16 code)
void PlaySoundFile()
void StopSoundFile()
void LaunchApp(TUint16* path)
void CloseApp(TUint16* name)
void StartKeyCapture()
void StopKeyCapture()
void ChangeProfile()
TInt ContinueChangeProfile(TAny* aObject)
TUint16* GetPath()
Instant-iates & Uses
CBTDiscoverer : public MSdpAgentNotifier, public MSdpAttributeValueVisitor
CSdpAgent* iAgent
SdpSearchPattern* iSdpSearchPattern
CSdpAttrIdMatchList* iMatchList
TUint iServicePort
TUint iAttrId
TUint16* iServiceName
RFileLogger* iLog
CBTServiceListContainer* iBtSlContainer
void ListServicesL(const TBTDevAddr& aAddress)
void PrintSDPError(TInt aError)
void NextRecordRequestComplete(TInt aError, TSdpServRecordHandle aHandle, TInt aTotalRecordsCount)
void AttributeRequestResult(TSdpServRecordHandle aHandle, TSdpAttributeID aAttrID, CSdpAttrValue* aAttrValue)
void AttributeRequestComplete(TSdpServRecordHandle aHandle, TInt aError)
void StartListL(CSdpAttrValueList& aList)
void VisitAttributeValueL(CSdpAttrValue& aValue, TSdpElementType aType)
void EndListL()
Instantiates & Uses
CBTServiceListContainer: public CCoeControl, public MEikListBoxObserver
CAknColumnListBox* iPortList
CCodeListener* iCodeListener
RFileLogger* iLog
CCodeListener* aCodeListener
void AddItemToList(TUint aServicePort, TUint16* aServiceName)
CBTServiceListContainer(RFileLogger* aLog, CCodeListener* aCodeListener)
void SetupScrollBarsL()
TKeyResponse OfferKeyEventL(const TKeyEvent& aKeyEvent, TEventCode aType)
void HandleListBoxEventL(CEikListBox* /*aListBox*/, TListBoxEvent aListBoxEvent)
Instantiates & Uses
Uses

Figure C.1: Class Diagram of iStuff Mobile smartphone background application. Only the relevant methods and member variables have been displayed.

C.1 Class CiStuffMobileAppUi

The class `CiStuffMobileAppUi` as the name suggests, is the application UI of the application. Regarding the iStuff Mobile application, this class has three main functionalities:

`CiStuffMobileAppUi` class is the UI of application

- Instantiate the `CiStuffMobileContainer` and the `CCodeListener` class instances.
- Layout the options which are visible to the user at bottom left and right side of the mobile phone screen. These options can be selected through left and right soft keys. The three options available at left soft key are "Connect To Proxy", "Disconnect From Proxy" and "Exit". The option available at right soft key is "Back". When the user selects "Connect" option `ConnectToServer()` method on `CCodeListener` instance is invoked, selecting "Disconnect" invokes `DisconnectFromServer()` method on `CCodeListener`, selecting "Exit" exits the application, and finally selecting "Back" invokes `SendToBackground()` method on `CiStuffMobileContainer` instance.
- Forward a key to the iStuff Mobile Proxy incase the user presses a key which application is in focus. Whenever a key is pressed the `HandleKeyEventL(const TKeyEvent& aKeyEvent,TEventCode aType)` method of the `CiStuffMobileAppUi` is invoked implicitly receiving the key code and the key type of the pressed key. This function then forwards the key by invoking `SendKeyToProxy(aKeyEvent.iCode,aType)` method on `CCodeListener` instance.

C.2 Class CiStuffMobileContainer

`CiStuffMobileContainer` class is a UI container which displays the main window of the application. This class has following two main functionalities regarding the implementation of iStuff Mobile background application:

This class is a UI container

- Load the iStuff Mobile bitmap image into the background of the container.

- Send the application to background when `SendToBackground()` method is invoked. This functionality is shown in code snip C.2.

```
TApaTask task(iEikonEnv->WsSession());

// Initialise the object with the window group id of
// our application (so that it represent our app)
task.SetWgId(CEikonEnv::Static()-
>RootWin().Identifier());

// Request window server to bring our application
// to foreground
task.SendToBackground();
```

Figure C.2: Code for sending the iStuff Mobile application to the background. These three lines are implementation of `SendToBackground()` method.

C.3 Class CCodeListener

`CCodeListener` implements the main functionality of iStuff Mobile

`CCodeListener` is the main class as far as the iStuff Mobile implementation is concerned. This class extends the Symbian `CActive` class which allows it be an active object and run in a cooperative threading environment (For more details on Active objects check Nokia Series 60 SDK documentation). Following are the main functionalities of this class:

`CCodeListener` allows the background application to connect to the iStuff Mobile Proxy over Bluetooth serial port

- Allow the application to connect to a Bluetooth device having Bluetooth serial port service. This action takes place when the user selects "Connect To Proxy" option from the left soft key menu. The code snip C.3 will bring up a Bluetooth station selection GUI. Once the user has selected a machine an object of `CBTServiceListContainer` class is constructed and displayed onto the GUI as top container. Then an object of `CBTDiscoverer` class is

```
TRequestStatus iLocalStatus;
User::LeaveIfError(iSocketServ.Connect());
User::LeaveIfError(iSocket.Open(iSocketServ, _L("RFCOMM")));

User::LeaveIfError(iDeviceSelector.Connect());
TBTDeviceSelectionParamsPckg selectionFilter;
TUUID serviceClass(0x1101); // SerialPort, uuid16: 0x1101,
       // see Bluetooth_11_Assigned_Numbers.pdf,
       // 4.4 Service Classes
selectionFilter().SetUUID(serviceClass);

iDeviceSelector.StartNotifierAndGetResponse(iLocalStatus,
KDeviceSelectionNotifierUid, selectionFilter, iResponse);
User::WaitForRequest(iLocalStatus);
```

Figure C.3: Code for bringing up the Bluetooth station search and selection GUI. `iSocketServer` is an instance of `RSocketServ` class, `iSocket` is an instace of `RSocket` class `iDeviceSelector` is an instance of class `RNotifier` and `iResponse` is an instance of class `TBTDeviceResponseParamsPckg`

constructed and `ListServicesL()` is invoked on it passing the Bluetooth address of the device selected. This would cause the `CBTDiscoverer` object to populate a list inside `CBTServiceListContainer` object with Bluetooth serial port services of the device. Finally when a user selects a service to connect to, code snip C.4 is invoked.

- Allow the application to disconnect from the proxy. This action takes place when the user selects ”Disconnect From Proxy” option from the left soft key menu which invokes the `DisconnectFromServer()` method of `CCodeListener`. The `DisconnectFromServer()` method simply closes the socket and cleans up.

- Listen to the incoming opcodes from the proxy and react accordingly. As soon as the mobile phone application is connected to the proxy, the `StartReceiving()` method of `CCodeListener` is invoked which checks if there is any opcode

`CCodeListener` listens to the incoming opcodes from the iStuff Mobile Proxy

```
TBTSockAddr address;
address.SetBTAddr(iResponse().BDAddr());
address.SetPort(portNo); //connect to the port number
selected by the user

iSocket.Connect(address, iLocalStatus);
User::WaitForRequest(iLocalStatus);
```

Figure C.4: Code for connecting to a particular service using the port number of that service. `portNo` is the port number of the service selected by the user.

from the proxy. If there is an opcode sent by the proxy the `RunL()` is invoked and the `data[0]` contains that opcode. The opcode is decoded in the `DecodeOpcode()` method and appropriate series of actions is initiated.

```
TPtrC Ptr(path);

CApaCommandLine * cmd = CApaCommandLine::NewL();
cmd->SetLibraryNameL(Ptr);    //set library name to the path
cmd->SetCommandL(EApaCommandRun);
TRAP(error, EikDll::StartAppL(*cmd)); //execute the command
```

Figure C.5: Code for launching an external application on the mobile phone. `path` variable contains the path of the application

`CCodeListener` enables launching of an external application

- Launch an external application. This action takes place when opcode `OPCODE_LAUNCHAPP` is received from the proxy. The background application expects the path length and the path string of the application to be launched to follow. After getting the path from the incoming stream, code snip C.5 shows how an external application is launched.

`CCodeListener` enables closing of an external application

- Close an external application. This action takes place when opcode `OPCODE_CLOSEAPP` is received from the proxy. The application expects the name length and name string of the application to be close to follow. The name should be the Caption of the appli-

```
TApaAppInfo AppInfo;
TApaTaskList aList(CEikonEnv::Static()->WsSession()); //get the
all tasks running on the phone

for(TInt i=0;i<AAppCount;i++){
    RSession.GetNextApp(AppInfo); //get info of the next
application
    TApaTask ATask3 = aList.FindApp(AppInfo.iUid); //find the
task in the task list
    if(ATask3.Exists()){
        if(AppInfo.iCaption.Find(Ptr) != KErrNotFound) {//if the
caption of the task is equal to the name
       ATask3.KillTask(); //kill the task
       killed = ETrue;
       break;}
    }
}
```

Figure C.6: Code for closing an external application on the mobile phone. `AAppCount` holds the number of active application on the mobile phone, `RSession` is an instance of `RApaLsSession` class, Ptr contains the name of the application received from the proxy.

cation to be close. After getting the name of the application to be closed, code snip C.6 shows how an external application is closed.

- Turn on the Backlight of the mobile phone. This action takes place when opcode `OPCODE_BACKLIGHT_ON` is received from the proxy. Backlight is switched on by invoking `User::ResetInactivityTime()` method.

- Simulate a key press on the current foreground application on the mobile phone. This action takes place when opcode `OPCODE_KEY_RECEIVED` is received from the proxy. After receiving this opcode, 6 more bytes and read from the serial port which are values of repeat, scancode and code respectively (See Also Section B.1—"iStuffMobileProxy class"). After converting these 6 bytes into the integers, code snip C.7 shows how a key is simulated on the foreground application.

`CCodeListener` enables simulation of key press on the foreground application

```
TKeyEvent event;
event.iRepeats = repeat;
event.iScanCode = scancode;
event.iCode = code;

TApaTask task(CCoeEnv::Static()->WsSession()); //get current window
session
task.SetWgId(CCoeEnv::Static()->WsSession().GetFocusWindowGroup()); //get
current foreground application

task.SendKey(event);  //send the key to the foreground application
```

Figure C.7: This piece of code shows how a key can be simulated on the foreground application of the mobile phone. Values of `repeat`, `scancode` and `code` are read from the incoming bluetooth serial port

`CCodeListener` allows changing of the ringing profile on mobile phone

- Change the ringing profile on the mobile phone. This action takes place when opcode `OPCODE_CHANGEPROFILE` is received from the proxy. The profile number follows this opcode and is read from the incoming bluetooth serial port. The profile number denotes the profile in the list of profiles displayed by the "Profiles" application on the mobile phone. To change a ringing profile, the background application launches the Symbian "Profiles" application, simulates key presses on the "Profiles" application to select the desired ringing profile and finally closes the application.

`CCodeListener` relays key presses to the Event Heap through iStuff Mobile Proxy

- Relay the user activity (key presses) from the mobile phone to the proxy. This action takes place when either user is pressing keys with iStuff Mobile application at the foreground or if the key capture is enables (Discussed in Section C.4—"Class CKeyListener"). In both cases `SendKeyToProxy(TUint16 code,TUint16 aType)` method of `CCodeListener` is invoked and this method relays the key to the proxy as shown in code snip C.8.

- Initiate key interception. This action takes place when opcode `OPCODE_START_KEYCAPTURE` is received from the proxy. The `CCodeListener` class in this case, simply creates an instance of `CKeyListener`

```
TBuf8<5> localData;

localData.Append(OPCODE_KEY_PRESSED); //send the opcode to the proxy
//send code and type of key as 4 bytes instead of 2 ints

localData.Append(code >> 8);
localData.Append(code);
localData.Append(aType >> 8);
localData.Append(aType);

iSocket.Write(localData,iLocalStatus);
```

Figure C.8: This code forwards a key from the mobile phone to the proxy. Conversion of `code` and `aType` variables from integers to bytes is also shown

class and invokes `StartL()` method on it.

- Stop key interception. This action takes place when opcode `OPCODE_START_KEYCAPTURE` is received from the proxy. In this case the `CCodeListener` invokes `StopL()` method on `CKeyListener` instance.
- Initiate sound playback. This action takes place when opcode `OPCODE_PLAYSOUND` is received from the proxy. After receiving this opcode, path length and path string of the sound file to be played is expected to follow. The `CCodeListener` class in this case, simply creates an instance of `CSoundPlayer` class passing path as constructor arguments.
- Stop sound playback. This action takes place when opcode `OPCODE_STOPSOUND` is received from the proxy. In this case the `CCodeListener` invokes `Stop()` method on `CSoundPlayer` instance.

C.4 Class CKeyListener

`CKeyListener` is responsible for intercepting key presses

`CKeyListener` class is responsible for intercepting user activity (key presses) no matter which application is running on the foreground. This class extends the Symbian `CActive` class which makes it an active object and allows it to run in a cooperative threading environment. Making

```
User::LeaveIfError(keyHandles[index] = wg->CaptureKey(EKeyBackspace, 0, 0)); //
the backspace key i.e. c
User::LeaveIfError(keyUADHandles[index++] = wg-
>CaptureKeyUpAndDowns(EStdKeyBackspace, 0, 0));
```

Figure C.9: Code showing how a window group can register to intercept a backspace key on the mobile phone (Other keys are registered in same matter). `wg` is an instance of class `RWindowGroup`

this class as an active object was necessary so that other classes of iStuff Mobile background application are active while keys are being intercepted.

`CKeyListener` registers to receive all key presses from the OS

The `CKeyListener` class intercepts the key presses by creating a new window session and registering it to receive all the key events from the mobile phone. Code snip C.9 shows how a single key is registered for interception.

Now, whenever user presses a key the `RunL()` method of `CKeyListener` object is implicitly invoked. Inside the `RunL()` method the key is retrieved from the window session and passed to the proxy by invoking `SendKeyToProxy()` method on a `CCodeListener` instance.

C.5 Class CSoundPlayer

`CSoundPlayer` class is responsible for playback and stopping of a sound file

`CSoundPlayer` class is responsible for playback of a sound file. This class extends the Symbian `MMdaAudioPlayerCallback` class to implement the sound playback. As soon as an object of `CSoundPlayer` class is constructed with path of the file as constructor arguments, an instance of class `CMdaAudioPlayerUtility`is created passing path as constructor arguments and `Play()` method is invoked on it. Invoking `Stop()` on `CMdaAudioPlayerUtility` object stops the playback of the sound file.

```
CTextListBoxModel* model = iPortList->Model(); //get the
most of the listbox
model->SetOwnershipType(ELbmOwnsItemArray);
CDesCArray* itemList =
STATIC_CAST(CDesCArray*,model->ItemTextArray()); //
get the list of item from the listbox
_LIT(KItem,"%d\t%s\t\t");
TBuf<256> item;
item.Format(KItem(),aServicePort,aServiceName);
itemList->AppendL(item); //add the item to the list
iPortList->HandleItemAdditionL(); //refresh action after
adding the item
```

Figure C.10: Code showing how service name and corresponding service port is added to the list. `iPortList` is an object of `CAknSingleNumberStyleListBox` class. `aServicePort` and `aServiceName` are received as parameters into this method

C.6 Class CBTServiceListContainer

`CBTServiceListContainer` is a secondary container for iStuff Mobile application which is constructed and displayed when user selects a device to connect to. This class consists a List which is populated by `CBTDiscoverer` class (Explained in next section). This container displays the list of service names and corresponding ports, and allows a user to select one of the services from the list. Once user has selected a service the container invokes `ConnectToService()` method on `CCodeListener` object passing the port number from the selected service as function arguments. Code snip C.10 shows how a service is added to the list dynamically.

`CBTServiceListContainer` displays all the services on a particular device and allows user to select one of them

```
switch(aType){
case ETypeUint: //if the type of the attribute value is
integer then it is port number

   if(iAttrId == 0x4)
      iServicePort = aValue.Uint(); //save the port number
break;

case ETypeString: //if the type of the attribute value is
string then it is port name

   if(iAttrId == 0x100){
      TInt len = aValue.Des().Length(); //legth of the
value
      iServiceName = new TUint16[len+1];
      for (int i = 0; i < len; i++){
         iServiceName[i] = aValue.Des()[i]; //save name
characters one by one
      }
      iServiceName[len] = '\0'; //string terminating
character
   }
break;

default:
break;
}
```

Figure C.11: Code showing how the bluetooth serial service name and port number is extracted from an attribute value of a service record.

C.7 Class CBTDiscoverer

`CBTDiscoverer` discovers all the BT serial port services on a machine

The `CBTDiscoverer` class is responsible for detecting services on a bluetooth serial port enabled station, extracting the service names and port numbers of serial port services and adding them to the list in `CBTServiceListContainer` instance. For this purpose `CBTDiscoverer` class extends the Symbian `MSdpAgentNotifier` and `MSdpAttributeValueVisitor` interfaces.

As soon as an object of `CBTDiscoverer` class is constructed, its `ListServicesL()` method is invoked with the bluetooth address of the device selected by the user. Inside the `ListServicesL()` method an instance of `CSdpAgent` class is constructed and `NextRecordRequestL()` method is invoked on it. Now when a record of a bluetooth serial port service is found on the device `NextRecordRequestComplete()` method of `CBTDiscoverer` is implicitly invoked.

To retrieve the attributes(port name and port number) of the service record, `AttributeRequestL()` method is called on `CSdpAgent` object. When a attribute of the service record is found, `AttributeRequestResult()` of `CBTDiscoverer` is invoked implicitly. Normally an attribute value is a complex structure. In order to retrieve the real port name and port number `AcceptVisitorL()` is called on `aAttrValue` which is the attribute value received.

Finally `VisitAttributeValueL()` method is implicitly invoked when a single value inside the complex attribute value structure is visited. Code snip C.11 shows how the port name and port number is saved when the attribute value is visited.

Service name and port number is added to a list on container

Once a single service record is finished `AttributeRequestComplete()` is implicitly invoked. This is where the port number and port name is added to the list inside `CBTServiceListContainer` and the next record is requested.

Bibliography

Gregory D. Abowd, Gillian R. Hayes, Giovanni Iachello, Julie A. Kientz, Shwetak N. Patel, Molly M. Stevens, and Khai N. Truong. Prototypes and paratypes: Designing mobile and ubiquitous computing applications. *Pervasive Computing, IEEE*, pages 67–73, 2005.

Apple Quartz Composer. URL `http://developer.apple.com/documentation/GraphicsImaging/Conceptual/QuartzComposer/qc_intro/chapter_1_section_1.html`.

Rafael Ballagas, Meredith Ringel, Maureen Stone, and Jan Borchers. istuff: A physical user interface toolkit for ubiquitous computing environments. In *Proceedings of the ACM CHI 2003 Conference on Human Factors in Computing Systems*, pages 537–544, Ft. Lauderdale, Florida, USA, April 2003.

Rafael Ballagas, Andy Szybalski, and Armando Fox. Patch panel: Enabling control-flow interoperability in ubicomp environments. In *PerCom 2004 Second IEEE International Conference on Pervasive Computing and Communications*, Orlando, Florida, USA, March 2004.

Rafael Ballagas, Michael Rohs, Jennifer Sheridan, and Jan Borchers. Sweep and point & shoot: Phonecam-based interactions for large public displays. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 1200–1203, New York, NY, USA, April 2005. ACM Press. ISBN 1-59593-002-7. doi: http://doi.acm.org/10.1145/1056808.1056876.

Rafael Ballagas, Faraz Memon, Rene Reiners, and Jan Borchers. istuff mobile: prototyping interactions for mobile phones in interactive spaces. In *Proc. PERMID, Work-*

shop on Pervasive Mobile Interaction Devices at PERVASIVE 2006, Dublin, Ireland, 2006a. LNCS.

Rafael Ballagas, Faraz Memon, Rene Reiners, and Jan Borchers. Rapidly prototyping mobile phone interactions in ubiquitous computing environments. In *Submitted to Ubicomp 2006*, Orange County, California, USA, 2006b.

Michael Beigl, Albert Krohn, Tobias Zimmer, Christian Decker, and Philip Robinson. AwareCon: Situation Aware Context Communication. In *Proceedings of Ubicomp 2003*, Seattle, USA, October 2003a.

Michael Beigl, Tobias Zimmer, Albert Krohn, Christian Decker, and Philip Robinson. Smart-Its – Communication and Sensing Technology for UbiComp Environments. ISSN 1432-7864 2003/2, University of Karlsruhe, 2003b.

Jacob T. Biehl and Brian P. Bailey. Aris: an interface for application relocation in an interactive space. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 107–116, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society. ISBN 1-56881-227-2.

CCC Cybelius Maestro. URL `http://www.cybelius.com/products/`.

H.-W. Gellersen, G. Kortuem, M. Beigl, and A. Schmidt. Physical Prototyping with Smart-Its. *IEEE Pervasive Computing Magazine*, 3(3):74–82, July–September 2004.

Saul Greenberg and Chester Fitchett. Phidgets: easy development of physical interfaces through physical widgets. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 209–218, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-438-X. doi: http://doi.acm.org/10.1145/502348.502388.

Beverly L. Harrison, Kenneth P. Fishkin, Anuj Gujar, Carlos Mochon, and Roy Want. Squeeze me, hold me, tilt me! an exploration of manipulative user interfaces. In *CHI '98: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 17–24, New York, NY, USA, 1998. ACM Press/Addison-Wesley Publishing Co. ISBN

0-201-30987-4. doi: http://doi.acm.org/10.1145/274644.274647.

Java Platform, Micro Edition. URL `http://java.sun.com/j2me/`.

Java Platform, Standard Edition. URL `http://java.sun.com/j2se/`.

Brad Johanson and Armando Fox. The event heap: A coordination infrastructure for interactive workspaces. In *WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, page 83, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1647-5.

Brad Johanson, Armando Fox, and Terry Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing*, 1(2): 67–74, 2002a. ISSN 1536-1268. doi: http://dx.doi.org/10.1109/MPRV.2002.1012339.

Brad Johanson, Greg Hutchins, Terry Winograd, and Maureen Stone. Pointright: experience with flexible input redirection in interactive workspaces. In *UIST '02: Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 227–234, New York, NY, USA, 2002b. ACM Press. ISBN 1-58113-488-6. doi: http://doi.acm.org/10.1145/571985.572019.

Scott R. Klemmer, Bjoern Hartmann, and Leila Takayama. d.tools: Integrated prototyping for physical interaction design. In *Stanford University Computer Science Technical Report*, 2005.

Johnny C. Lee, Daniel Avrahami, Scott E. Hudson, Jodi Forlizzi, Paul H. Dietz, and Darren Leigh. The calder toolkit: wired and wireless components for rapidly prototyping interactive devices. In *DIS '04: Proceedings of the 2004 conference on Designing interactive systems*, pages 167–175, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-787-7. doi: http://doi.acm.org/10.1145/1013115.1013139.

Macromedia Flash. URL `http://www.macromedia.com/software/flash/flashpro/`.

Macromedia Flash Lite. URL `http://www.macromedia.com/software/flashlite/`.

Microsoft PowerPoint. URL `www.microsoft.com/powerpoint/`.

Jakob Nielsen. Iterative user-interface design. *Computer*, 26(11):32–41, 1993. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.241424.

Nokia Series 60. URL `http://www.forum.nokia.com/main/0,6566,010_400,00.html`.

OPI Artistic License. URL `http://www.opensource.org/licenses/artistic-license.php`.

Trevor Pering, Rafael Ballagas, and Roy Want. Spontaneous marriages of mobile devices and interactive spaces. *Commun. ACM*, 48(9):53–59, 2005. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1081992.1082020.

Python for Series 60. URL `http://www.forum.nokia.com/python`.

Mika Raento, Antti Oulasvirta, Renaud Petit, and Hannu Toivonen. Contextphone: A prototyping platform for context-aware mobile applications. *IEEE Pervasive Computing*, 4(2):51–59, 2005. ISSN 1536-1268. doi: http://dx.doi.org/10.1109/MPRV.2005.29.

René Reiners. The Patch Panel GUI: A Graphical Development Environment For Rapid Prototyping Of Physical User Interfaces For Ubicomp Environments. *Diploma Thesis*, 2006. URL `http://www-i10.informatik.rwth-aachen.de/reiners.html`.

Michael Rohs and Beat Gfeller. Using camera-equipped mobile phones for interacting with real-world objects. In Alois Ferscha, Horst Hoertner, and Gabriele Kotsis, editors, *Advances in Pervasive Computing*, pages 265–271, Vienna, Austria, April 2004. Austrian Computer Society (OCG). ISBN 3-85403-176-9.

Salling Clicker. URL `http://www.salling.com/`.

Albrecht Schmidt, Kofi Asante Aidoo, Antti Takaluoma, Urpo Tuomela, Kristof Van Laerhoven, and Walter Van de Velde. Advanced interaction in context. In *HUC '99:*

Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing, pages 89–101, London, UK, 1999. Springer-Verlag. ISBN 3-540-66550-1.

Symbian Series 60. URL `http://www.s60.com/`.

Technology for Enabling Awareness. URL `http://www.teco.edu/tea`.

Teleo. URL `http://www.makingthings.com/teleo.htm`.

Daniel Wigdor and Ravin Balakrishnan. Tilttext: using tilt for text input to mobile phones. In *UIST '03: Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 81–90, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-636-6. doi: http://doi.acm.org/10.1145/964696.964705.

Index

Typeset May 16, 2006