Pocket PC Code Structure

This document explains the software structure of the Pocket PC component of the Real-time Systems sensor networks project. The whole environment is available via subversion and forms a single EVC++ workspace.

Hardware requirements

The course uses HP IPAQ 4150, 4350, and 5550 Pocket PCs. Other Pocket PCs may work fine, but are untested. We assume the following hardware is available:

- 400 MHz Intel XScale or similar ARM processor
- 64 MB of memory, ideally with >32 MB as "program memory"
- Veo Traveler 130 SDIO camera, or equivalent.
- Wifi (802.11b) networking
- Bluetooth networking

Software requirements

The target operating system is Windows Mobile 2003. Earlier versions of Windows Mobile may work, but they are not tested. Beyond this, the following software is assumed.

- Microsoft Embedded Visual C++ 4.0 SP4.
- Microsoft Windows Mobile 2003 SDK.
- Highpoint BTAccess Bluetooth software development kit. This is necessary to program the Widcomm Bluetooth stack in this machines. It is an affordable alternative to the absurdly overpriced Widcomm SDK.
- Veo camera SDK for the Veo Traveler 130.

Libraries, components, and applications

The software is divided into libraries, components, and applications. What is meant by component here is different from, say, an "active x" component. The idea here is that libraries are passive code that only executes due to an API call. Components on the other hand are active – when they are initialized, they launch one or more internal threads that execute in parallel with the thread that initialized them.

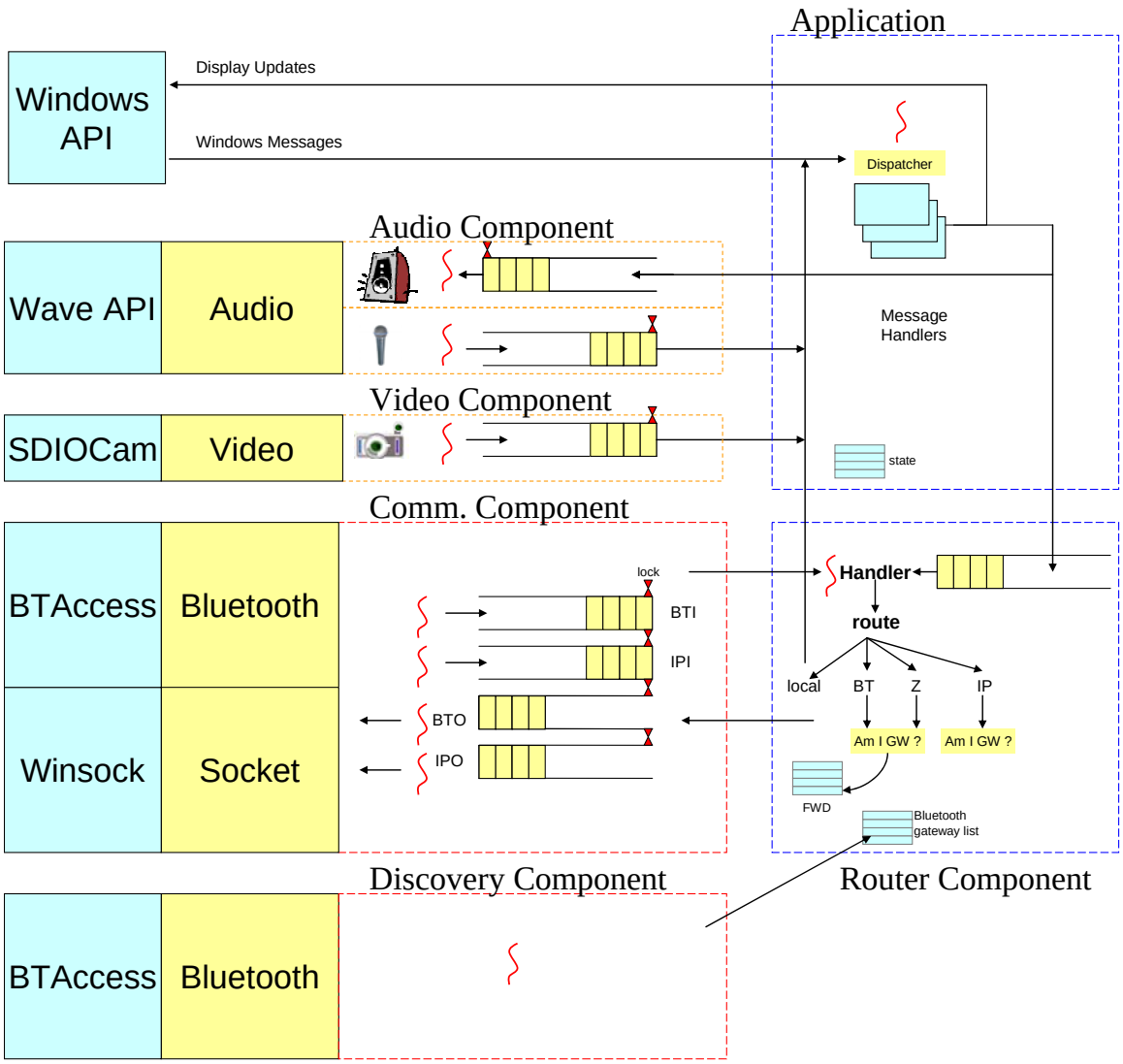
From the programmer's perspective, both libraries and components look like static libraries that can be called from C++. The libraries generally have simple C-style APIs.

In some cases, it is necessary for a library to use an application window as a windows message target. In these cases, the expectation is that the corresponding window procedure will pass the message back to the library. All of the libraries include simple macros which can be trivially added to a window procedure's switch statement.

Components may explicitly pass data back to the application as a windows message. Hence, component initialization requires a handle to a window. The convention here is

that component's header file indicates which messages types are delivered. All messages include a pointer to some structure as their LPARAM parameter.

The following illustrates a typical integration of libraries, components, and application code to create an application:



On the left are the underlying libraries or APIs that the PPC code builds on (core Windows API, Wave API for audio input/output, SDIOCam for video input, BTAccess for Bluetooth communication, and Winsock for IP communication). The next layer are the libraries supplied to the student. These provide a greatly simplified API, hiding as many irrelevant details of the underlying libraries as possible.

The dotted rectangles are the components. The squiggly lines represent threads within the components. Each of these threads can have its priority manipulated. Most of the

components also contain one or more message queues. The ordering of these queues can also be manipulated by the student.

At the upper right is the application itself. It is responsible for initializing the components and maintaining the user interface. At component startup, it passes the component a handle to a window that the component will be able to send messages to. The messages that the components send are documented in the component's header file.

Message library

Communication between PPCs, PPCs and Motes, and between components within a PPC are through messages that are strictly defined in a message class hierarchy. Every message class is a struct, meaning all of its fields are directly accessible. The intent is to make it straightforward to use these classes within C as well as C++. Examine code\ppc\libs\messages\messages.h for more details on messages.

Messages need to be serialized to/from buffers before they can be sent on the network. There is a global serialize routine and a global unserialize routine. It should not be necessary for students to modify the per-class serializers. In the buffer, a message is stored as a depth first traversal, packed, with basic types in network byte order. The message is preceded by a header (0xdeadbeef), and a length (4 bytes). The very next 4 byte integer is a message type tag which is used to dispatch the appropriate unserializer.

A detailed description of the messages, roles, and actions are given in the design documents (labs\design_docs*)

Audio library

The audio library implements sound input and output using the Windows Wave API. It simplifies the interface considerably and provides double-buffering to give glitch-free playback and recording provided that buffers are supplied to it at a high enough rate. You can find out more about the audio library in code\ppc\libs\audio\audio.h and in the example application code\ppc\apps\example_audio_socks.

Video library

This library provides a simple interface on top of the SDIOCam API. Unlike the audio library, the user of the video library must forward internal windows messages to the library and must also receive explicit messages from the library that indicate when new frames are ready. The frames that arrive, from either picture or video mode are in a structure that is intentionally compatible with the video message class in the message library. For more information on the video library, see code\ppc\libs\video\video.h and code\ppc\apps\example_video_socks.

Sockets library

This is a fairly large library designed to simplify programming with Berkeley sockets. There are only small portions that the student will need to use since all communication is unicast UDP. For more information, see code\ppc\libs\sockets\socks.h. The audio

example application (code\ppc\apps\example_audio_socks) is an excellent one to see how this library will be used by the students.

Bluetooth library

This library provides a simplified interface to the BTAccess library for programming the Bluetooth communication stack. Note that on the Pocket PC, only a single incoming and a single outgoing connection can be simultaneously extant. Like the video library, the user must forward Bluetooth-related windows messages to the library. Unlike the video library, no user-level messages are sent by the library. To examine the library, look at code\ppc\libs\bluetooth and for an example of how to use it, see code\ppc\apps\example_bluetooth.

Audio component

The audio component supports both audio input and output. When initialized a notification window handle must be given. The component includes two threads, whose priority can be manipulated independently and externally To play sound, start audio output and then simply pass audio messages to the component. The record sound, start audio input. The component will then send your notification window messages which have audio messages attached. There are two queues that can be managed. See code\ppc\components\audio

Video component

The video component operates identically to the audio component except that video messages are passed in and out. Passing a video message in causes it to be immediately drawn on the notification window. Only a single thread exists in the video component. There is a single queue that can be managed. See code\ppc\components\video.

Communication component

This component moves messages to and from the IP and Bluetooth networks and the local machine. It's the only component that needs to use message serialization since all other message handoffs are internal to the machine. Unlike the audio and video components, this component uses a pull model instead of a push model for locally destined messages. This means that the user must request messages. No user-visible windows messages are exchanged. All four threads can have their priority manipulated. There are four queues that can be managed. See code\ppc\components\commqueue

Router component

This component is responsible for routing every message that arrives at the machine, regardless of its source. If a message is locally destined, it will deliver it to the notification window using a special message type. Thus, unlike the communication component, the router component is push-based. See code\ppc\components\router

Discovery component

Because of the nature of the Bluetooth protocol, it is necessary to occasionally discover new devices. This component does this. It has a single thread. Both the thread priority

and the discovery rate can be set. When a new device is discovered, a message is sent to a notification window with the address of the device. Typically, this should be passed to the route component which will send the gateway request and keep track of which devices it is gatewaying for.