KTH Stockholm

Touchbutton & Software

KTH Stockholm Karsten Becker kbec@kth.se http://karsten-becker.de/

Touchbuttons

Goals

The goal is to define a way the user can interact with the newly created platform. One way is to have some sort of button. But because this is a research project we explored several possibilities that could be deployed into the wrist band.

Existing technologies

Technology	Functionality	Advantage
Resistive	Small gap between conductive layers	Position can be exactly measured by measurement of the resistance
Capacitive	Distortion of electric field using finger	Virtually no contact to the material required
Touch button	Finger closes gap using skin resistance	Very easy to implement, close to GSR

All technologies are very suitable for straight material. But as this is deployed to a wristband it won't be straight.

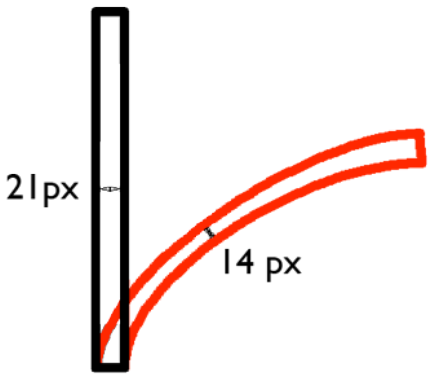
Bending the technologies

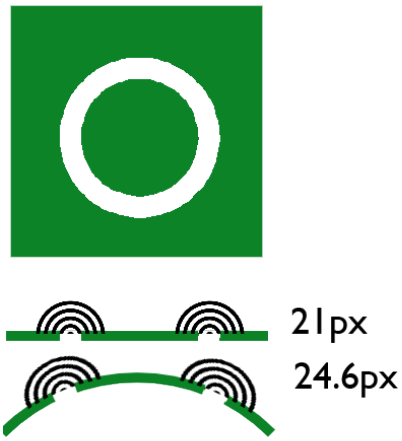
When bended some of the properties of the technologies change and can render them unusable.

Resistive

The resistive touch screens that are used in mobile phones feature an extremely small gap between the two layers of conductive material. Usually they are filled with air.

The graphic to the right shows what happens when the material is bend. The size of the gap decreases and false touch registrations are very likely.





Capacitive

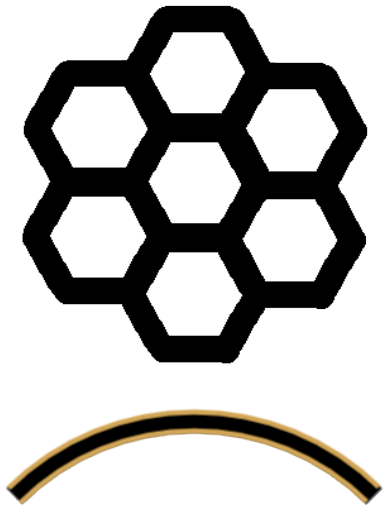
Capacitive touch buttons are composed of 2 parts. A electrical positive inner surface surrounded by a GND plane. In this Gap an electrical field builds up. When the user touches the button, or even just comes close to it, the field is distorted. The distortion can be used to trigger a button.

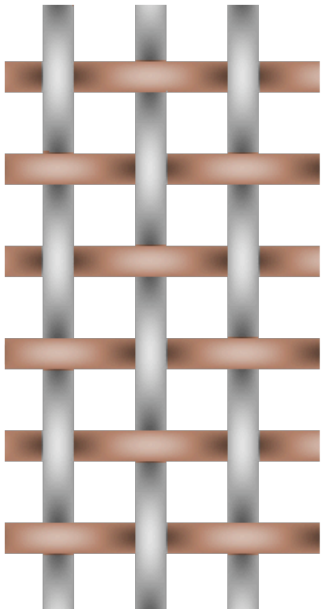
However the right graphic shows that bending the material varies the size of the gap and thus influences the field. This makes it harder to distinguish a touch from a bend.

Honeycomb touch button

The idea of the honeycomb touch button is very similar to the resistive measurement. Except that the information about resistance is discarded. Also the gap is not filled with air, but with a honeycomb material. This material has gaps which allows the upper layer to contact the lower layer when it is compressed.

The properties when bend are better than the resistive material as the gap size could be increased.





Touch thread

Touch thread is an evolution of the touch button where VCC and button lines are at a close distance to each other. When the finger touches the material it shortens those 2 lanes and a very small current can be measured. This current can easily be used to create a logical 1 or 0.

Bending the material does only have a negligible influence on the functionality. It is very easy and cheap to produce.

Summary of the properties

Some technologies are easy to realize and are immune to bending.

Technology	Bendable	Production
Resistive	Not with small gap	Difficult, must be bought
Capacitive	Will influence field	Hard to produce
Honeycomb	Yes	If material available, easy
Touch thread	Yes	Easy

From the summary it is obvious why we focused on the last 2.

Prototyping

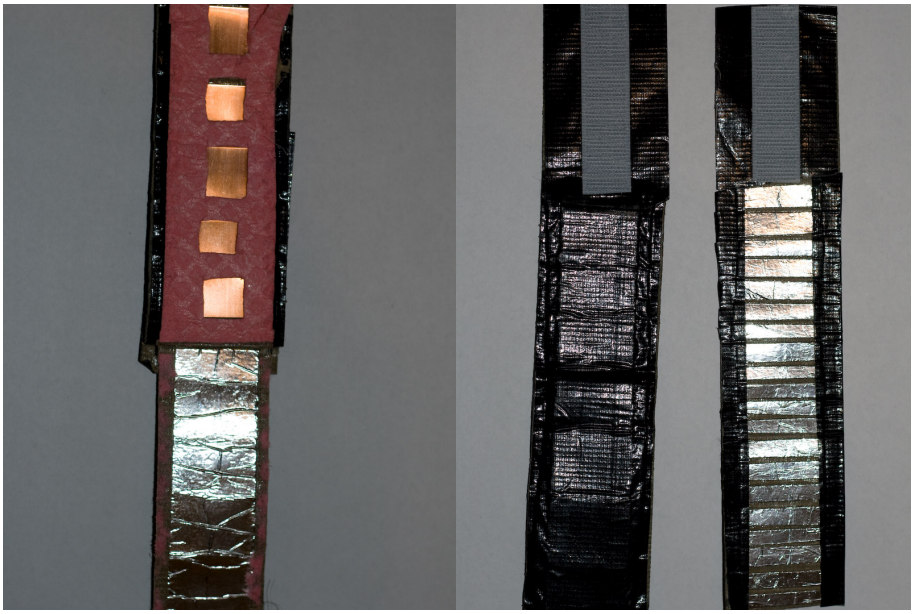
The most difficult part for creating a prototype was to find the honeycomb material. Luckily we found some at Clas Ohlsen.

A sponge has all the required properties, but is lacking holes. Those can easily be created using a knife or alike.

So the required tools for creating prototypes are:

- Duct tape
- Valcro tape
- Aluminum tape
- Copper plate
- Sponge
- Cardboard

Using those materials a number of prototypes have been created.

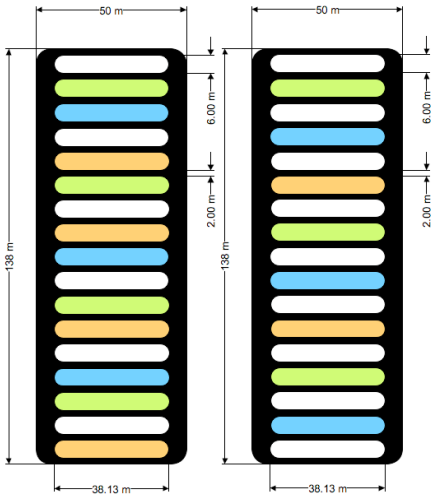


Results from the first prototyping

The honeycomb material prototype has one severe problem. The holes are easy to detect and need quite a lot of compression in order to be functional. A sliding across the buttons was not as easily possible as with the prototype to the right. We thus focused on creating a more advanced version of it.

Wiring the buttons

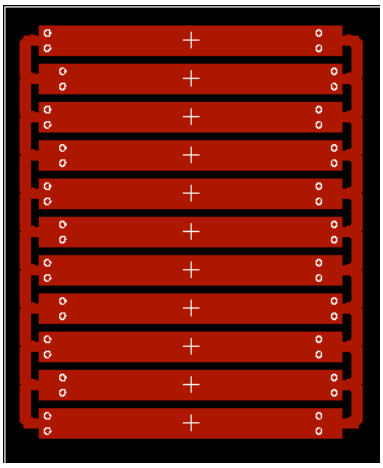
In theory it would be very nice to connect every button. However this creates a lot of wiring and also lots of electronic parts. In order to reduce the complexity we identified that a number of 3 distinguishable buttons would be sufficient to detect a direction.



Each color identifies one of the 3 buttons. White represents VCC. There are 2 practical arrangements for those buttons. The left version has less white buttons and thus a higher density of colored buttons. However it also has dead spots between colors. The right version was thus chosen to be more practical.

The first hand soldered version looked promising, but it also took a long time to produce. With the use of the milling machine we were able to produce smaller button areas, equal spacing among all buttons and a more appealing look.

Milling the buttons



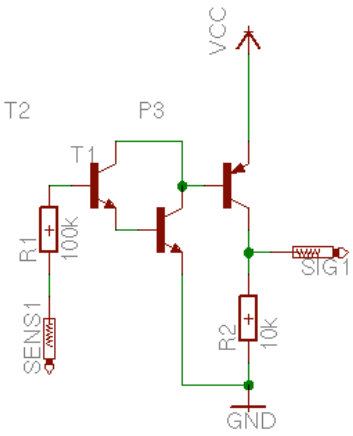
In the first attempts to mill the buttons the metal parts connecting on the side were too small and broke. The buttons broke lose and were pulled into the vacuum cleaner.

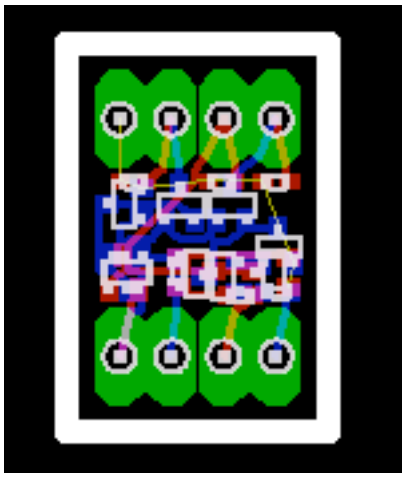
With wider traces keeping the buttons in place and some double sided tape the buttons could finally be kept in place. Also the traces on the side could be used for supplying the VCC to every second field, thus dramatically reducing the amount of soldering.

The easiest way to solder those buttons is to use a blank via-wire and solder it onto each button of the same color. Then isolate it to prevent it from touching the other fields.

The other hardware

The hardware for detecting the touch is really simple. It consist of one 100k resistor that limits the amount of current in case buttons are shorted. One pull down resistor of 10k. And a darlington pair transistor, or as in our case 2 chained transistors.





Results for touch buttons

After identifying the touch button solution it proved to work very solidly. And there are lots of possible additions. For example when a cheap power supply is used field effects can be seen. Those could be used to sense touch-less “touches”.

One of the prototypes contained plastic materials to see how the PCB parts would influence the wearing comfort. And it turned out to be a big deal. Thus creating a PCB that is as small as possible is really crucial. Also some more thought has to be spent on how the buttons are actually placed on the final wrist band.

Software

Goals

During the communication with our project partner it became clear that a easy to use yet flexible software architecture should be built. One of the areas of researches requires to try different timings. Also the whole board should be able to run for a long time from battery.

With this in mind a solution has been built upon a domain specific language (DSL). A DSL is able to describe a certain process in a language that is known to people that work in this domain. Unlike a general purpose programming language it is not designed to describe every program possible. The restricted number of possibilities increases the readability significantly. If well designed a person not familiar with any programming knowledge should be able to create “programs” that do something useful.

The sequence description DSL

The sequence description language contains 2 parts. The first part contains the declaration of modules. The second one contains sequences. Those can be declared in different files.

Modules

A module defines an abstract piece of hardware that can be used in sequences.

```
module Red_LED OUT {}
```

This declares a module with the name Red_LED. There exist 4 kinds of modules:

- OUT - used for modules that can only be switched on/off
- PIN/ADC - used for modules that can sample something
- UART - used for modules that can send data

```
module Bluetooth UART{
      parameter heartbeatOnly;
      parameter gsrOnly;
      parameter sendAll;
      parameter sendAlive;
      flag dataReceived;
}
```

This declares a module Bluetooth that can communicate. For sending data it has 4 freely named parameter that can be passed to the send method. It also has a flag that can signal the arrival of data.

Sequences

A sequence describe several events that are triggered after each other. Each sequence is evaluated every ms. There are 3 kinds of sequences:

- ONCE - only executes once
- REPEAT - when the end of the sequence is reached it starts from the beginning
- IMMEDIATE - gets executed only once and may not have a wait statement. This is for sending short commands to the processor. Unlike ONCE and REPEAT they are executed immediately.

```
sequence seq_yellow REPEAT {
      switch_on Yellow_LED;
      wait 753 ms;
      switch_off Yellow_LED;
      wait 123 ms;
}
```

This sequence executes at the following times:

time [ms]	commands evaluated	meaning
0	`switch_on Yellow_LED;` `wait 753 ms;`	Switch on the yellow LED, wait for 753 ms
753	`switch_off Yellow_LED;` `wait 123 ms;`	Switch off the yellow LED, wait for 123 ms
876	`switch_on Yellow_LED;` `wait 753 ms;`	Switch on the yellow LED, wait for 753 ms
...	...	...

The following commands are supported:

Command	Argument [Type]	Description
`switch_on`	Module [All]	calls the switch_on_Module() method
`switch_off`	Module [All]	calls the switch_off_Module() method
`sample`	Module [PIN/ADC]	calls the sample_Module() method
`send`	Module [UART] (parameter)	calls the send_Module(int) method with optional value of parameter. If no parameter is given 0 is passed.
`wait (or)`	amount unit (or flag)	waits for the amount of time. Unit can be one of (`ms,s,m,h`). The or is optional. If a flag is specified the sequence will restart from beginning if the time is passed. If the flag is raised before the time expired the sequence is continued.
`seq_on`	Sequence [ONCE, REPEAT]	Activates a sequence. As immediate sequences can not be stored they can not be activated again.
`seq_off`	Sequence [ONCE, REPEAT]	Deactivates a sequence. As immediate sequences can not be stored they can not be deactivated again.

Examples

```
sequence seq_acc REPEAT {
      switch_on Accelerometer;
      sample Accelerometer;
      switch_off Accelerometer;
}
```

This example will switch on the accelerometer every 1 ms and take a sample. After that it is switched off to save energy.

```
sequence init ONCE {
      //switch_on Bluetooth;
      //switch_on Proximity;
      switch_on Touchsensor;
      switch_on GSR;
}
```

This sequence switches on some hardware. After that the sequence automatically deactivates itself.

```
sequence seq_sendData REPEAT {
      seq_off seq_heartbeat;
      switch_on Red_LED;
      switch_on Bluetooth;
      wait 132 s or dataReceived;
      send Bluetooth gsrOnly;
      switch_off Bluetooth;
      switch_off Red_LED;
      seq_on seq_heartbeat;
      wait 185 s;
}
```

This sequence is a little more powerful. At first the heartbeat sequence is deactivated. As you might not want disturb the transfer of data. Then the red LED is switched on to indicate an ongoing transfer. The Bluetooth module is switched on and waits for a hello signal for 132s. When that is received the GSR data is send. After that the Bluetooth and Red LED modules are switched on again. Also the sequence for measuring the heartbeat is switched on again. After 185 more seconds this sequence will repeat itself.

Creating the DSL

In order to realize a DSL for this project OpenArchitectureWare XText is used. This is a tooling for creating textual DSLs. The procedure is quite simple and automatically generates an fully features Eclipse Editor plus tooling to generate code from the DSL.

At first you specify your programming language as EBNF in a file of the type xtxt.

The EBNF for this language looks like this:

```
Body:
     (imported=Import)?
     (modules+=Module)*
     (sequences+=Sequence)*;
Import:
     "import" uri=URI ";";
Module:
     "module" name=ID type=ModuleType "{"
          (parameters+=Parameter)*
          (flags+=Flag)*
     "}";
Flag:
     "flag" name=ID ";";
Parameter:
     "parameter" name=ID ";";
Enum ModuleType:
     adc="ADC" | pin="PIN" | uart="UART" | out="OUT";
Sequence:
     "sequence" name=ID type=SequenceType "{"
          (actions+=Action)*
     "}";
Action:
     (wait=Wait)|
     (modAction=ModAction)|
     (send=Send) |
     (toggle=Toggle);
Toggle:
     toggle=ToggleType sequence=[Sequence] ";";
Enum ToggleType:
     seq_on="seq_on" | seq_off="seq_off";
ModAction:
     type=ModActionType module=[Module] ";";
Enum ModActionType:
     swon="switch_on" | swoff="switch_off" | sample="sample";
Send:
     "send" module=[Module] (parameter=[Parameter])? ";";
Wait:
     "wait" time=Time (or=OrCondition)?";";
OrCondition:
     "or" flag=[Flag];
Enum SequenceType:
     repeat="REPEAT" | immediate="IMMEDIATE" | once="ONCE";
Time:
     amount=INT unit=TUnit;
Enum TUnit:
     ms="ms" | s="s" | m="m" | h="h";
```

From this EBNF several files/projects are generated. One editor for eclipse with syntax highlight, outlines, navigation etc.. And another project that is for generating the code from the sequence description. Those are contained in .XPT files.

Creating the bytecode

From the DSL several files are generated. Among them some Java files that contain the bytecode, constants and helper functions.

The sequence.java contains the constants for the bytecode. Here an excerpt:

```
//Declarations for module: Accelerometer

/**
 * Switches on the module Accelerometer.
 * <br>bytecode: (byte)0x0A
 */
public static final byte SWITCH_ON_ACCELEROMETER=(byte)0x0A;

/**
 * Switches off the module Accelerometer.
 * <br>bytecode: (byte)0x0B
 */
public static final byte SWITCH_OFF_ACCELEROMETER=(byte)0x0B;

/**
 * Collects a sample for the module Accelerometer.
 * <br>bytecode: (byte)0x0C
 */
public static final byte SAMPLE_ACCELEROMETER=(byte)0x0C;
```

Sequences.java contains the byte code for each generated sequence. Here an excerpt:

```
public static final byte seq_seq_yellow[]=new byte[]{
      Sequence.SWITCH_ON_YELLOW_LED,
      Sequence.WAIT, (byte)0xf0,(byte)0x2, //752 ms
      Sequence.SWITCH_OFF_YELLOW_LED,
      Sequence.WAIT, (byte)0x7a,(byte)0x0, //122 ms
};
```

SequenceWriter.java conrtains helper methods to write the data to some OutputStream. Each of those files is generated when the generator is invoked.

Conclusion

The DSL allows an easy modification of the sequences. The bytecode generation allows to change the behavior of the code even during runtime. And the interpreter is very lightweight. So all goals of the software design have been reached. For further documentation I will create a Wiki which can keep up with the development of the Sequence IDE.