

Association of Mobile Devices through Visual Codes

Semester Project

June 1, 2005

Supervisor: Michael Rohs

Patrick Jayet

jayetp@student.ethz.ch

The goal of this semester project is the development of an application that enables the association between a smartphone and a display through a visual communication. It focuses in particular on the transmission of meta-data through a sequence of 2-dimensional barcodes and its technical realisation in Java on the sender's side and Symbian C++ on the receiver's side.

Contents

1. Introduction	7
1.1. Visual Code System	7
1.2. Task	7
1.3. Motivation	7
1.3.1. Privacy	7
1.3.2. Communication Context	8
1.3.3. Data Capacity	8
1.4. Starting Point	8
1.4.1. On the Sender	8
1.4.2. On the Receiver	9
2. Design	10
2.1. Overall Process	10
2.2. Packet Format	10
2.2.1. Version 1: One Code per Frame	10
2.2.2. Version 4: Four Codes per Frame	11
2.3. Application Level Checksum	13
3. Implementation	14
3.1. From ANSI C++ to Symbian C++	14
3.1.1. Basic Types	14
3.1.2. Naming Conventions	14
3.1.3. Inheritance	15
3.1.4. Memory Management	15
3.1.5. Two-Phase Construction	15
3.2. Architecture	16
3.2.1. Sender	16
3.2.2. Receiver	18
4. Performance	21
4.1. Frame Interval	21
4.2. Sequence Decoding Time	22
4.2.1. One Code per Frame	22
4.2.2. Four Codes per Frame	23
4.2.3. Throughput Comparison	25
4.3. Frame Interval on a Faster Device	25
5. Related Work	27
6. Conclusion	28
A. Using the Emulator for Debugging	29
A.1. General Consideration	29
A.2. Video Sequences as Camera Input	29

- B. Raw Measurements** **32**
- B.1. Frame Interval 32
- B.2. Sequence Decoding Time 34
- B.3. Frame Interval on a Faster Device 43

- C. Hardware and Software Details** **45**
- C.1. Smartphone 45
- C.2. Software 45

1. Introduction

1.1. Visual Code System

A visual code system [Roh] has been developed in the Distributed Systems Group at the ETH. It enables to read and decode a 2-dimensional matrix code using a mobile phone and its embedded camera. No additional infrastructure is needed. Visual codes can be printed on paper or on electronic displays like those of mobile phones, PDAs and large displays.

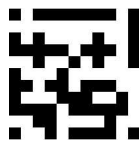


Figure 1.1.: A visual code

1.2. Task

A communication initiated through a visual code in a smart environment has advantages in regard to privacy. The address of a mobile device doesn't need to be communicated in the first phase of a communication process. The communication phase using radio frequency—Bluetooth or wireless LAN—is dependent on the anonymous meta-data transmitted in the first phase and of an acknowledgement from the user.

The data capacity of a single visual code is limited to 76 bits¹. If a larger capacity is needed, it is possible to use a sequence of visual codes or to display more than one code at the same time. It is also possible to combine the two techniques—display a sequence of more than one code in each frame. In that way a larger data quantity can be transmitted.

For applications where privacy is a major concern, a complete communication through visual transmission is also possible. In that case, the two devices need to be equipped with a camera and a display respectively. A communication in each direction is done by displaying a code or sequence of codes on one device and recording it with the camera by the other device. However, in that project just a one-way communication has been studied.

1.3. Motivation

1.3.1. Privacy

In the world of ubiquitous computing it becomes important to pay attention to *privacy*. On the one hand can a networked and smart environment be extremely useful. A number of services are ready to be used. Plenty of devices stay interconnected, each one dedicated to a specific task. On the other hand one's privacy becomes hard to protect. Each communication

¹The raw data capacity of a code is 83 bits and an encoding using a [83,76,3] linear code provides an effective capacity of 76 bits.

between two devices leaves a trace in the digital world, be it a MAC or an IP address recorded somewhere in a logfile, be it a session number stored as a cookie. It becomes important to offer communication paradigms that guarantee some degree of anonymity.

Let's consider a service for mobile device, for example in a public location. It is desirable that the service provides a way of querying it, of getting meta-data about it, anonymously. These meta-data could be pricing information or some kind of description about it. The goal would be not to give information about one's identity in the preliminary phase—when meta-data about a service are gathered, but only when the actual transaction takes place.

1.3.2. Communication Context

Another important issue is that a radio based communication does not permit to identify one's partner. That is particularly true in a smart environment with several wireless enabled devices. A communication through visual codes represents an initial context to the subsequent communication. The visual channel guarantees that the link is established with the device we are seeing and not another one [McCu04]. The visual code or a sequence of them can be printed on a sticker located on the device to identify or directly on a display. An active attack on the visual tag is possible but relatively difficult without being detected. Some kind of visual authentication can be performed with the device. In addition, the physical address can be transmitted by the visual channel and used for the communication initiation, as for example in [Sco05]. That can have several advantages like assuring that the communication is initiated with the desired device and reducing the manipulation of the user to select a device in a list of available ones.

1.3.3. Data Capacity

A visual code alone enables to transmit 76 bits of data. For the transmission of meta-data or the setup of an encrypted or authenticated link based on a visual communication, the mentioned data capacity is too limited.

A possible approach to solve that issue is to show more than one code at a time, another one is to cycle through a sequence of codes, as mentioned in section 1.2. This enables to increase of several order of magnitude the data capacity of the visual channel and allow the development of several interesting applications needing a larger bandwidth.

1.4. Starting Point

As we are going to mention quite often the two sequence versions described above, let's define the version 1 as a sequence of 1 code per frame and the version 4 as a sequence of 4 codes per frame.

1.4.1. On the Sender

The starting point at the sender's side, on the computer, is composed of a few Java classes that can display a single visual code. There is also a class `Enc` that has some static methods to perform the encoding of a 76 bits code using a [83,76,3] linear code².

²A [n,k,d] linear code represents a code of length n containing 2^k code words of minimum Hamming distance d from each other. Each code word has an effective payload of k bits.

1.4.2. On the Receiver

On the smartphone, there exists already a set of classes, implemented in Symbian C++, for the decoding of visual codes inside an image. In addition to that, a small test application has already been programmed, which receives a sequence of predetermined 32 bits numbers. That test application will be the starting point of the project on the receiver.

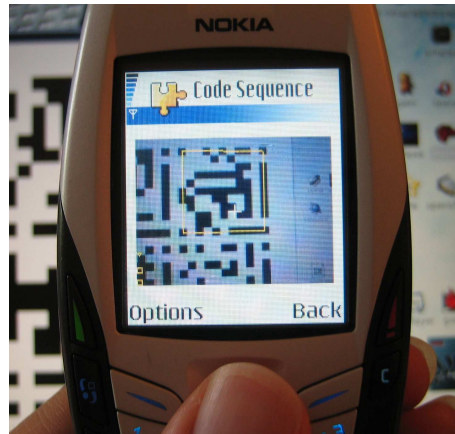


Figure 1.2.: The sequence receiver on the smartphone

2. Design

2.1. Overall Process

A sequence of visual codes is a sequence of 1 to 16 frames composed of 1 or 4 codes, corresponding to the version 1 or 4 as defined above. A basic setup of the transmission is illustrated in figure 2.1. Each code is shown on a display for an interval time Δt . After having reached the last frame n , the process starts again and frame 1 is shown. During that time the other device records the frames and extracts the data from them.

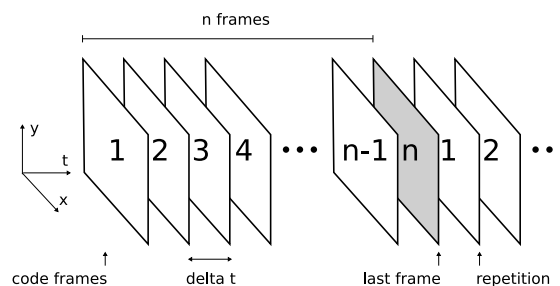


Figure 2.1.: Basic setup at the sender's side. The code frames are shown on a display repetitively from 1 to n .

The overall communication scheme is depicted in figure 2.2. On the sender side the input data is split in chunks. A header, containing some administrative information—the numbering of the frames, of the codes and some other information—is appended. The 76 bits of a packet are encoded using a $[83,76,3]$ linear code, which enables the detection of two flipped bits on a received code¹. The code is stored in a buffer and the sender's module alternatively shows each frame during an interval time Δt on the display.

At the receiver's side the camera records the frames, which are processed and from which the codes are extracted and decoded. According to the information from their header, they are stored in a buffer of the right type and at the right position. When the sequence is complete, the data is taken out of the buffer, stripped from the header, put together and delivered to the client method through a call-back.

2.2. Packet Format

2.2.1. Version 1: One Code per Frame

The figure 2.3 gives an overview of the format for the version 1—one code per frame. Let's have a look at the different fields:

- **v field** A first important information that needs to be stored is the code version, 1 or 4 codes per frame. It is coded in the v field as one bit—0 for 1 code and 1 for 4 codes.

¹The $[83,76,3]$ code has a minimal distance δ_{min} of 3.

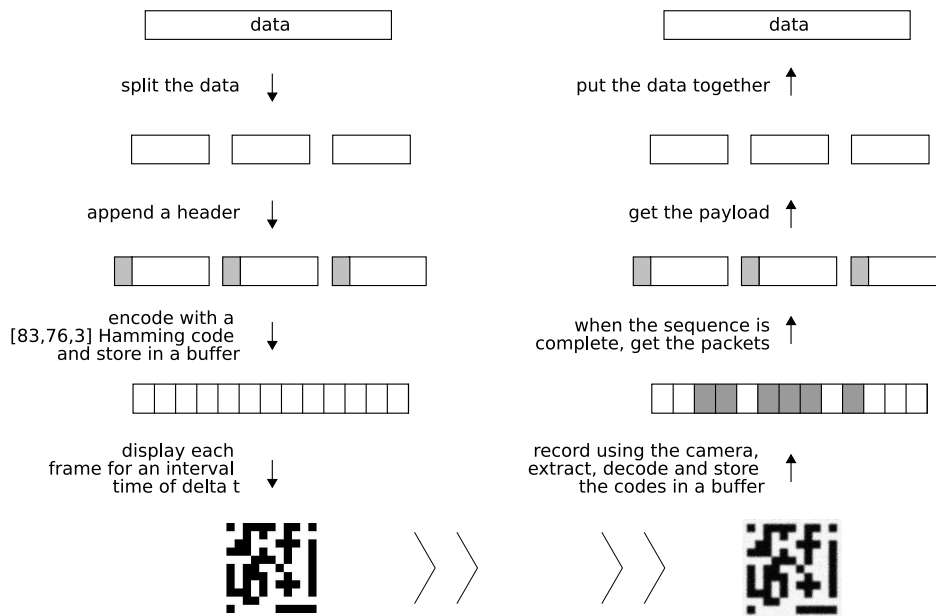


Figure 2.2.: Illustration of the overall process. First on the sender's side, the data is split, a header is appended, the packet is encoded using a $[83,76,3]$ linear code and stored in a buffer. Each frame is finally displayed on the screen for an interval time Δt . On the receiver's side, the camera record an image, the visual codes are extracted, the packets are decoded from the $[83,76,3]$ linear codes and stored in a buffer. When the sequence is complete, the packets are taken, the header stripped out and the data put together.

- **seq nb field** Another important information is the frame numbering. It is stored as a sequence number in the *seq nb* field, of a length of 5 bits. The reason to use 5 bits is that although sequence lengths of up to 16 frames have been used in that project, future extension of the system should be possible without having to change the implementation.
- **lt field** The information that a frame is the last one needs to be stored. This is done by the *lt* field set to 1 in case a frame is the last one, and 0 otherwise.
- **payload field** Finally the data payload, in the *payload* field, occupies the remaining part of the packet.
- **len last field** Additionally, the header of the last frame contains a length field, *len last*, indicating the length in bits of the payload for the last frame. The payload length can be up to 63 bits and a field of 6 bits is needed. The length in bits of the overall sequence can be calculated using the *last* bit and the length of the last frame.

The data capacity of a sequence of 16 frames for the version 1 is calculated as follow:

$$C_1 = 15 \times 69 + 63 = 1098 \text{ bits } (\simeq 137 \text{ bytes})$$

2.2.2. Version 4: Four Codes per Frame

For the version 4, using 4 codes per frame, the packet formats are a bit more complicated. Let's define a naming convention for the 4 codes inside a frame. They are named from A to D

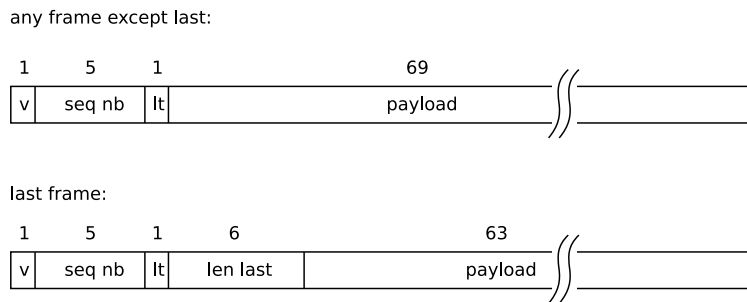


Figure 2.3.: Packet format for the version 1, using 1 code per frame

D as illustrated in figure 2.4.

The header of version 4 is essentially similar to the version 1. The code A contains a last lt field indicating that the whole frame is the last one. That field is not necessary in the codes B to D as the receiver has to wait until all codes of all frames have arrived in order to process the data.

The format of the last frame is also a bit different: the last frame code A contains a $len\ last$ field, as in the version 1, but that field represent the length of data for the whole last frame, from code A to D . Please refer to the figure 2.5 for an overview of the format.

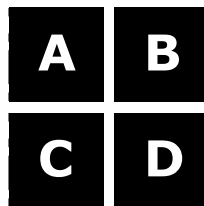


Figure 2.4.: Four codes together as used in a version 4 frame.

The supplementary or modified fields of the version 4 are the following:

- **cnb field** An additional information that needs to be stored in the header is a code number, situating a code inside a frame². It is the function of the cnb field, coded on 2 bits—00 for code A to 11 for code D .
- **len last field** In the version 4, the $len\ last$ field is larger, because there is up to $3 \times 68 + 58 = 262$ bits in the last frame. Thus $len\ last$ has a length of 9 bits.

As we can notice from the figure 2.5, each code of a frame has its own sequence number. The reason for that is the following. If it is always sure that the receiver can decode the 4 codes of a frame, this redundancy would be unnecessary. But the decoding of each code has a probability smaller than 1, as we will see in the performance section. If the sequence number is just present on the first code—code A —and for some reason it is not decoded, we would be unable to know to which frame belong the other 3 codes and to store them in the buffer at the right position. That would be undesirable. The solution presented here has the advantage of being more robust.

In addition, that redundancy permits to make an additional check. In case an arriving code has an erroneous sequence number, some kind of majority vote can be applied in order to

²As every code in a frame has the same sequence number.

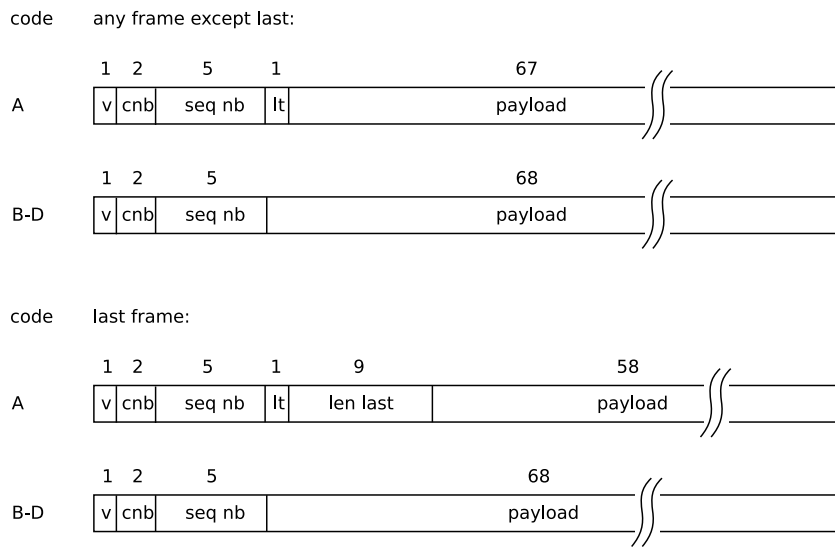


Figure 2.5.: Packet format for the version 4, using 4 codes per frame

throw it away. The rule that is used here is to throw a code away if there are 2 or more other codes with another and equal sequence number in the same frame. Of course, this kind of protection is of limited scope. This enables to detect only errors in a sequence number and not in other parts of the header or in the payload. A more robust solution would be to use some other encoding scheme, more resistant to errors than the [83,76,3] linear code—for example a [83,68] error correcting Reed-Solomon code, carrying 68 bits of payload and 15 bits of redundancy, as explained in [McCu04] on pages 15-16.

Similar to the version 1, the maximum capacity for a sequence of 16 frames, this time with 4 codes per frame, can be calculated as:

$$C_4 = 3 \times 16 \times 68 + 15 \times 67 + 58 = 4327 \text{ bits } (\simeq 540 \text{ bytes})$$

Please refer to the figure 2.5 for an overview of the different formats and payload lengths for the version 4.

2.3. Application Level Checksum

At the application level, a 16 bits CRC checksum is prepended in front of the data to transmit. That enables a verification at the receiver's side, where a CRC checksum is built on the received data and compared to the prepended CRC checksum.

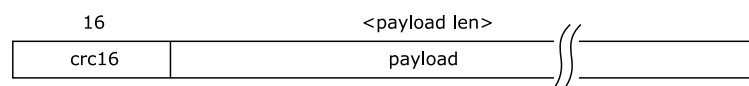


Figure 2.6.: Packet format at the application level. A 16 bits CRC checksum is prepended to the payload. At the receiver's side, a CRC of the data is also calculated and compared to this one to check for data integrity.

3. Implementation

3.1. From ANSI C++ to Symbian C++

Symbian C++ is not completely compatible with ANSI C++. There are some differences that come either from efficiency consideration of Symbian C++ or from different design decisions having been taken—ANSI C++ was completely defined as a standard only in 1998. Let's have a look at some relevant aspects. Please refer to [SymC++] for an exhaustive treatment.

3.1.1. Basic Types

The Symbian C++ dialect doesn't support the basic ANSI C++ types. It defines its own basic types. Some of them are listed in the table 3.1.

Table 3.1.: Overview of the basic types in Symbian C++

ANSI C++ type	Symbian C++ type
byte	TInt8
short int	TInt16
int	TInt or TInt32
unsigned int	TUInt or TUInt32
float	TReal32
double	TReal or TReal64
bool	TBool
void*	TAny*

3.1.2. Naming Conventions

The following naming conventions apply:

- **C Classes** are heap-allocated classes that can allocate memory resources. They are derived from the CBase class and have a name beginning with a C like CFoo.
- **T Classes** don't own any external objects. Example of these are some basic types like TInt or TReal.
- **R Classes** contain handles or pointers on external resources allocated elsewhere. For example an array of pointers as defined in RPointerArray<type> just holds references on the resources stored in it.
- **M Classes** are abstract or interface classes. As there is no interface in C++, abstract classes are used instead.

3.1.3. Inheritance

The inheritance scheme of Symbian C++ has been simplified in regard to the usual inheritance scheme of C++, partly because of efficiency considerations. It is actually not far from the model used in Java. A C class can be derived from 1 or more M classes but just from 1 C class. That corresponds to a single inheritance of actual classes and multiple inheritance of interface or abstract classes. The parent C class must be the first in the list in the definition of the child class.

A few other restrictions apply and can be found in [SymC++].

3.1.4. Memory Management

As in ANSI C++, there is no garbage collection or automatic memory management. On that point, Symbian C++ differs from programming languages like Java or C#. Special care must be taken to delete manually any instantiated objects.

3.1.5. Two-Phase Construction

A situation that may arise is the following. A new object is allocated but there is not enough memory for that operation. In Symbian C++, an overloaded new operator, new (ELeave) triggers an exception, called a *leave* in the Symbian jargon, in such a case. That has the advantage of not returning partly allocated objects.

For some reason, a constructor in Symbian C++ is not allowed to allocate objects. Let's see why. A wrong constructor like that

```
class CFoo: public CBase{
    // ..

    TInt iVal;
    CSomeClass* obj;

    // wrong!
    CFoo(){
        iVal = 10;
        obj = new (ELeave) CSomeClass();
    }
}

// an object of CFoo is allocated
CFoo* o = new (ELeave) CFoo(); // (*)
```

would be problematic. If the system at line (*) is able to allocate memory for the object o, but runs out of memory when executing the constructor CFoo as it is allocating memory for the member obj, then the constructor would leave and the already allocated resources would remain allocated. That would represent a memory leak.

A right way to construct such a class is to make a distinction between the part of the constructor that may not leave and the part that may leave, and separate them. A stack called the `CleanupStack` then holds a reference on the object that may leave, for the second phase of the construction in `ConstructL`:

```

class CFoo: public CBase{
    // ..

    TInt iVal;
    CSomeClass* obj;

    CFoo(){
        iVal = 10;
    }

    ConstructL(){
        obj = new (ELeave) CSomeClass();
    }
}

// the class is then created using the cleanup stack
CFoo* o = new (ELeave) CFoo();
CleanupStack::PushL(o);
o->ConstructL();
CleanupStack::Pop();

```

We see that the constructor is separated in two. CFoo affects a value to iVal and ConstructL allocates some resource for obj. If the constructor ConstructL leaves, the system can look at the elements that are still on the cleanup stack and free the corresponding resources.

It is also possible to encapsulate that construction mechanism in two methods, NewL and NewLC. We let the interested reader find further information on that topic and on some Symbian C++ specific trapping mechanisms in [SymC++].

3.2. Architecture

3.2.1. Sender

The sender program is implemented in Java. An UML scheme of it can be found in figure 3.2. The SeqGenerator class is the main class. It is a wrapper for the SeqGeneratorUI class and can be used in two ways:

- First to send some specific data. In that case the parameters to use are:

```
SeqGenerator data <data> <mode> [<delta t>]
```

where <data> is the data to send in decimal or hexadecimal—in that case being prefixed with 0x—and <mode> corresponds to 1 for the version 1 and 2 for the version 4. <delta t> is optional and represents the frame interval in ms; its default value is 250 ms.

- Second to generate a given amount of codes with random data. In that case the parameters to use are:

```
SeqGenerator test <code nb> <mode> [<delta t>]
```

where `n` is the number of codes, not of frames, to generate. That means that for the version 4—with 4 codes per frame—to transmit n frames, we have to specify $4n$ codes. The reason to do this is that, in some case, we want only a part of the codes of the last frame to be used. We can specify a number of $4(n - 1) + a$ codes, where $a = 1..4$ to have only a codes occupied on the last frame. On the receiver's side the sequence will be detected as complete and delivered when all codes of the first $n - 1$ frames plus the necessary a codes of the last frame n have arrived.

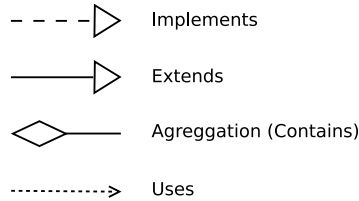


Figure 3.1.: Definition of the relations used in the UML schemes

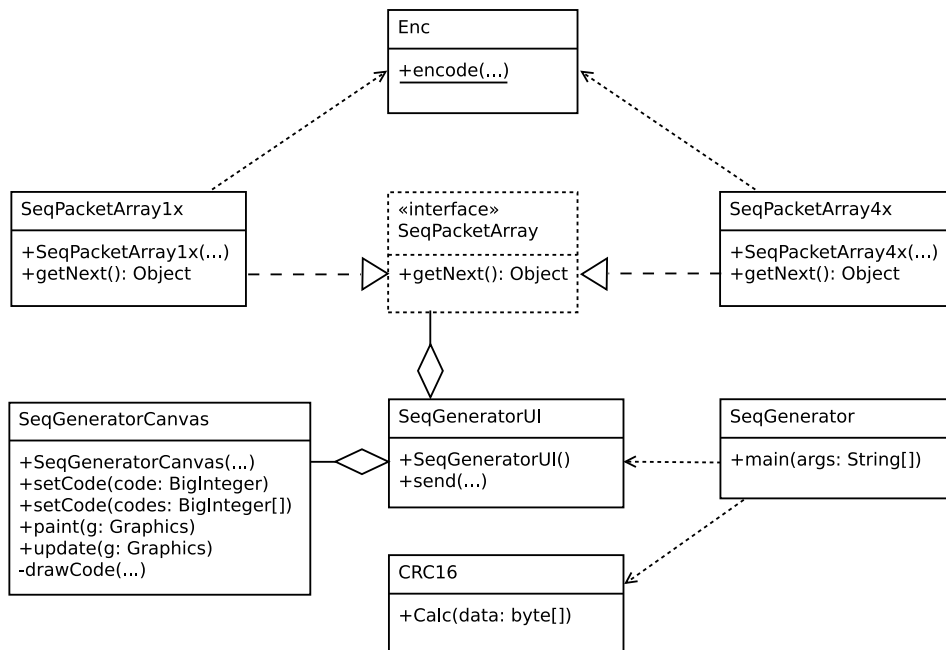


Figure 3.2.: UML scheme of the sender module, on the computer side, implemented in Java.

Let's have a look at the details of the different classes of the sender's module:

- The `SeqGenerator` class calculates a 16 bits application level CRC16 checksum, to verify the data integrity at the receiver's side. The data format at the application level has already been discussed in section 2.3. The `SeqGenerator` class then creates an instance of the `SeqGeneratorUI` class and sends the data using its `send` method.
- The `CRC16` class just contains a static method `Calc` to calculate a 16 bits CRC checksum.

- Another important class is `SeqGeneratorUI`. It extends `javax.swing.JFrame`. Its constructor `SeqGeneratorUI` calls its parent constructor to open a new window. Its `send` method takes as parameter an array of bytes for the data, the data length in bits, and the mode—1 for version 1, 2 for version 4. The `send` method creates a new `SeqPacketArray` of the right type and a `SeqGeneratorCanvas`. In its main loop it invokes the `getNext` method of `SeqPacketArray` to get the next packet data from the buffer, introduces it in the `SeqGeneratorCanvas` object using the `setCode` method, calls its `repaint` method and finally sleeps for a time interval Δt .
- `SeqGeneratorArray1x` and `SeqGeneratorArray4x` are wrapper classes for a packet buffer. They both implement the `SeqPacketArray` interface. The constructor takes as parameter the data as `BigInteger` and its length in bits. It splits the data in smaller chunks, appends a header to them, encodes it in a [83,76,3] linear code using the `Enc` class and stores the result in an `ArrayList`. The `getNext` method then delivers each time the next packet from an iterator corresponding to that array.
- The `Enc` class contains a static overloaded method `encode` that encodes its input using a [83,76,3] linear code.
- `SeqGeneratorCanvas` takes care of the low-level drawing procedures. An overloaded `setCode` method sets the current code or set of codes of the object. A `paint` method takes care of the drawing of the codes and calls for each code a private `drawCode` method for that purpose. The `update` method actually just calls `paint` and is needed because the class is an extension of the `java.awt.Canvas` class.
- Apart from that there are also two exceptions defined, the `InvalidModeException` and `DataOverflowException`. They handle the corresponding cases.

3.2.2. Receiver

The receiver module, on the Symbian smartphone series 60¹, is implemented in Symbian C++.

The UML scheme of the module is represented in figure 3.3.

Let's consider the classes of the receiver's module in some details:

- `CCodeSequencePacket1x` and `CCodeSequencePacket4x` are wrapper classes for what we call a packet, a received code. The two classes extend the abstract class `MCodeSequencePacket`. A constructor `ConstructL` is provided, that takes a pointer to a `CBigInteger` as argument, that represents the data of that code or packet. Then different methods are provided to get the values of specific fields like the version, the sequence number, the last bit or the payload length. For the version 4x, there is a method to get the code number inside the frame and another for the payload length of the whole last frame.
- `CCodeSequencePacketArray1x` and `CCodeSequencePacketArray4x` are also wrapper classes, this time for a buffer or array that stores the set of incoming packets. Here also, they extend an abstract class called `MCodeSequencePacketArray`. A first method is `Add`. It takes as parameter an array of `CCodeInfo`, containing the decoded codes and referenced from a `RPointerArray` object. That `Add` method creates a new `CCodeSequencePacketXx` for each incoming code and enters them at the right position in a buffer. A second method is `GetData`. It fetches and returns the data of the

¹Details on the hardware of the smartphone are provided in appendix C.

whole buffer. Another one is `IsComplete` to query the completeness of the incoming data sequence. There are also some getter methods like `GetDataLen`, `GetFrameNb`, `GetPacketMap` or `GetType`.

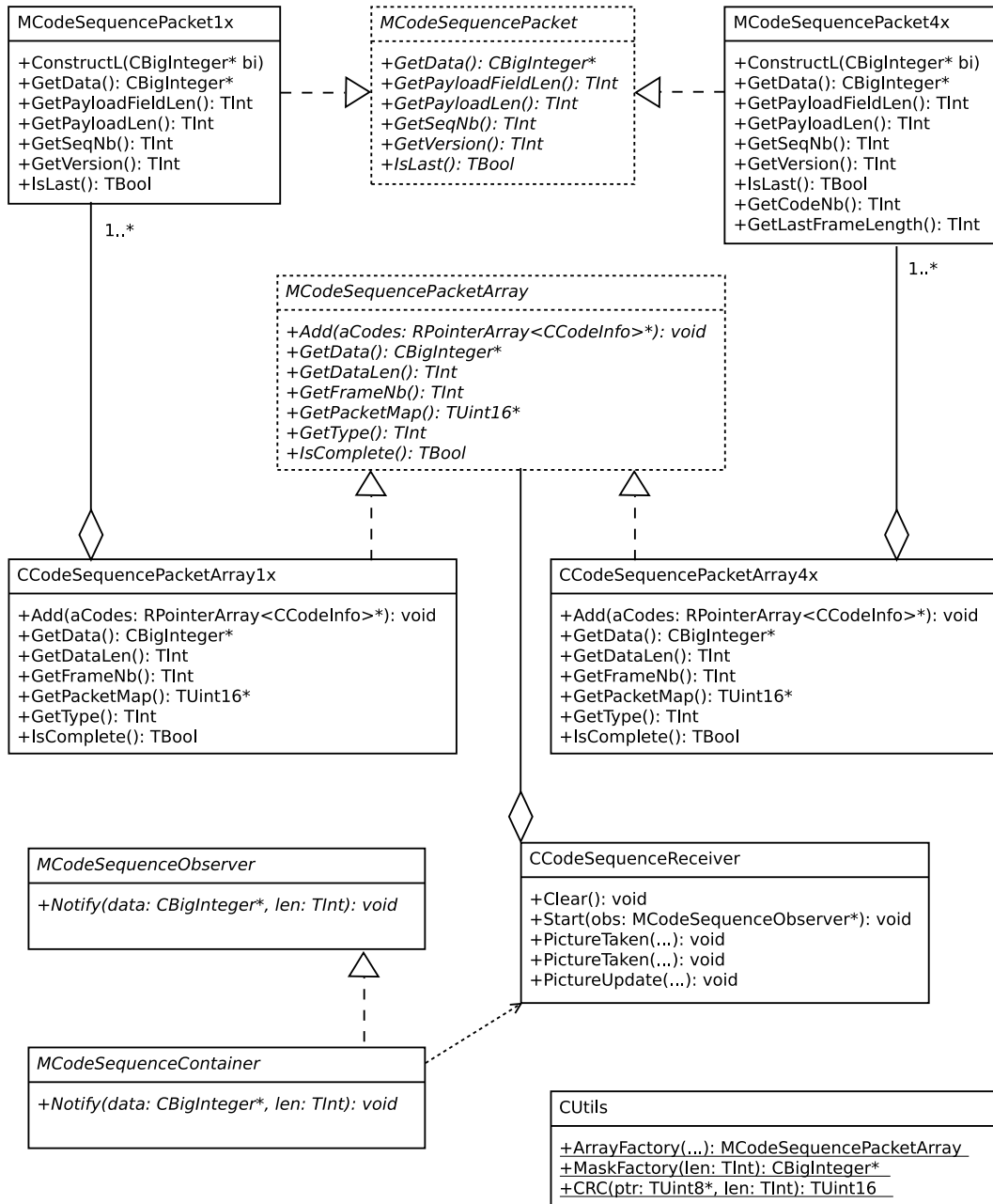


Figure 3.3.: UML scheme of the receiver's module, on the smartphone, implemented in Symbian C++.

- An important class is `CCodeSequenceReceiver`. It has a constructor taking different parameters representing the graphical context, as it is going to use it. It extends the abstract class `MVisualCodeObserver` as it creates a `CVisualCodeSystem` object and registers itself for a call back—each time the visual code system decodes a frame, a method `PictureUpdate` is called. `Start` and `Clear` enable to start or reinitialize the

receiver.

- The class that uses the receiver, as a client, is in that case `CCodeSequenceContainer`. It extends the `MCodeSequenceObserver` abstract class, defining a `Notify` class for the call back. When the whole sequence is decoded, that method is called in order to deliver the data.
- Finally there is `CUtils`. This class provides different static methods. `ArrayFactory` is a factory method that returns a packet array of the right type given a set of codes. `MaskFactory` takes as parameter a length and returns a mask of the given length. `CRC` calculates a CRC code of some data.

4. Performance

Three series of tests have been done on the code sequence mechanism. The standard device used for the tests, unless otherwise specified, is a Nokia 6600 smartphone¹. The first test has the goal to determine the optimal frame period Δt during which a code frame is shown on the display. Optimal means here that we want the mean decoded time of a predefined sequence of codes to be as small as possible.

The second series of tests has the goal to determine the mean decoding time for sequences of different lengths, in optimal conditions, and for the two different modes, version 1 and version 4. Optimal conditions means here that we are interested in reproducing the optimal conditions a user is able to reach. The mobile phone is almost still, the code size as seen by the phone camera is large.

Finally the third series aims at testing the system on a newer and faster smartphone, a Nokia 6630. As for the first series, an optimization of the frame interval time is performed.

4.1. Frame Interval

As explained in the previous paragraph, the goal of this test is to determine the optimal frame interval Δt for our standard testing smartphone, a Nokia 6600.

In a first phase, the setup of the test is to measure the mean decoding time of a sequence of 16 frames, in version 1—1 code per frame—for different values of the frame interval Δt . Values of Δt from 0.1 s to 0.35 s have been taken, 5 measurements per interval time. The results along with the standard deviation can be seen in figure 4.1(a).

The best decoding time is reached for an interval Δt of 0.20 s and 0.25 s, 0.20 being a bit better than 0.25. However the standard deviation is still quite large and it is not clear which one of the two is better.

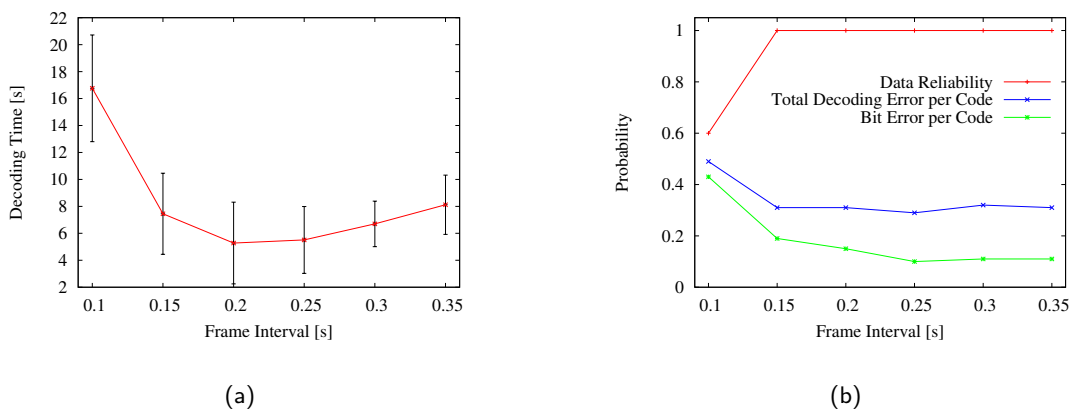


Figure 4.1.: Graphic showing the mean decoding time of a sequence of 16 frames in function of various interval times. The best results seem to be achieved for an interval time Δt of 0.20 and 0.25 seconds.

¹Details about that device can be found in appendix C.

Because the standard deviation for these two values of Δt is large in regard to the mean decoding time, a second series of tests between these two interval times has been made. This time 20 measures for each value of Δt have been taken. The result is shown in figure 4.2. The mean decoding time for the interval of 0.25 s, 7.6 s, is a bit smaller than the value for 0.20 s, 8.3 s. Thus, the interval time of 0.25 s has been chosen as the optimal interval time. We can notice however, that the standard deviation for these two values of Δt are quite large, what has also been noticed during the measurements. Large variations from one decoding time to the other have been observed. Therefore an exact optimization of that parameter seems difficult. However I think that a value of 0.25 s is a reasonable option.

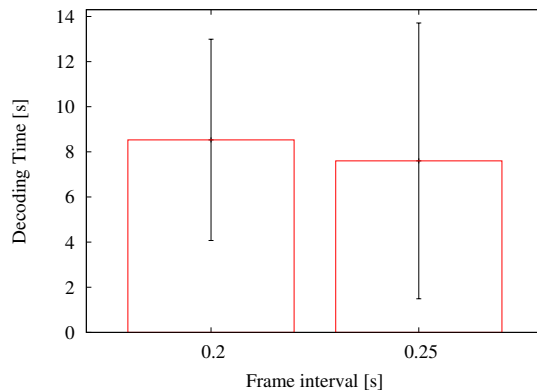


Figure 4.2.: An additional set of 20 measures for each of the two interval time Δt of 0.20 and 0.25 s have been carried out. As we see, the decoding time for Δt of 0.25 s seems slightly better though the standard deviation with respect to the decoding time is still large.

4.2. Sequence Decoding Time

4.2.1. One Code per Frame

A series of measurements has been carried out in order to get an estimation of the sequence decoding time for different lengths, in optimal conditions. Optimal conditions mean that the mobile phone is not moving itself, that the code image is taken such that it appears large for the mobile phone camera and the plane of the mobile phone is parallel to the plane of the display. For each sequence length from 2 to 16 frames, 10 measurements have been taken. For a sequence of just 1 frame, only 1 measurement has been made, as the result is expected to be exactly the same—decoding time of 0 s. The mean decoding times and their standard deviations are represented on figure 4.3(a) as well as the minimal theoretical decoding time and a least square fit of the sample.

On the one hand, we can see that the least square fit is quite good in respect to the minimal theoretical decoding time. The slope of the least square fit is 0.318 and the theoretical one 0.250 so the least square fit is just about 30% worse. On the other hand, for a sequence of more than 10 frames, we begin to observe huge variation between *good* and *bad* sequences, what we see on the figure with an increasing standard deviation. For example for a length of 11 frames, the best decoding time is 2.5 s and the worst 7.9 s. There is more than a factor 3 between the two values, what can become problematic in terms of usability. However, until a length of about 10, the standard deviation remains in an acceptable range.

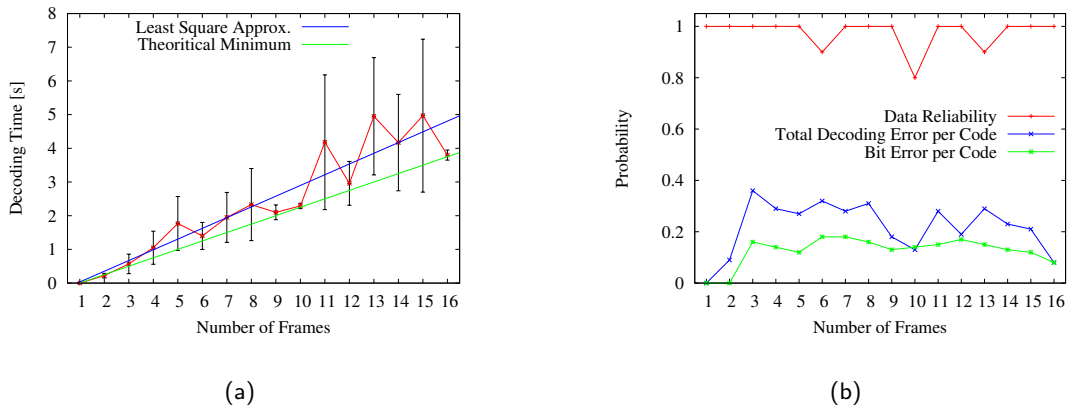


Figure 4.3.: (a) Mean decoding time for sequences of 1 to 16 frames, version 1 with 1 code per frame, and its standard deviation. The theoretical minimum as well as a least square fit is also represented. (b) The data reliability, total decoding error and bit error for the corresponding sequence lengths.

Some other parameters have been represented in figure 4.3(b): the data reliability, tested through the CRC checksum at the application level, the bit error rate—proportion of codes that have been recognized as having one or two bit errors by the [83,76,3] linear code, and the total decoding error per code, taking the hypothesis that every image frame on the mobile phone contains ideally one decoded code. Here we can see that the data reliability remains good although not perfect. Possibly one or at most two codes out of 10 contain erroneous data. The total decoding error is larger than the bit error per code, what is normal. On all sequences the mean of total decoding error is 0.22 and the mean of bit error is 0.13.

4.2.2. Four Codes per Frame

The test setup is similar to the version with 1 code per frame. The figure 4.4(a) represents the mean decoding time, its standard deviation as well as the theoretical minimum and a least square fit of the sample.

On the one hand, we can notice on the graph 4.4(a) that the least square fit is no longer close to the theoretical minimum, as for the version 1. The slope of the least square fit is this time 0.957. It represents 383% of the theoretical minimum. On the other hand it is clear that we have here about 4 times more data per frame than in the version 1, so we can hardly expect the same range of decoding time. In addition we can notice that sequence lengths greater than 8 begin to show large standard deviation. For example for a length of 9 frames, the best decoding time is 4.4 s and the worst 15.5 s. That situation is even worse for a length of 13 frames where the best decoding time is 5.1 s and the worst 40.9 s. It is actually not very surprising as the codes have to be smaller, 4 together must fit on the phone camera image with the same resolution. In addition, because of the wide angle of the mobile phone camera, some codes are possibly distorted if the camera isn't placed exactly at the center of the image. Another factor that explains these large variations is that, it can happen, as for the length of 13 frames, that almost all codes are decoded except of one or two and that the remaining code(s) repeatedly, for some reason, cannot be decoded on successive sequence sweeps.

The next graph, on the figure 4.4(b) is similar to the graph for the version 1. We see the data reliability, the total decoding error as well as the bit error probability.

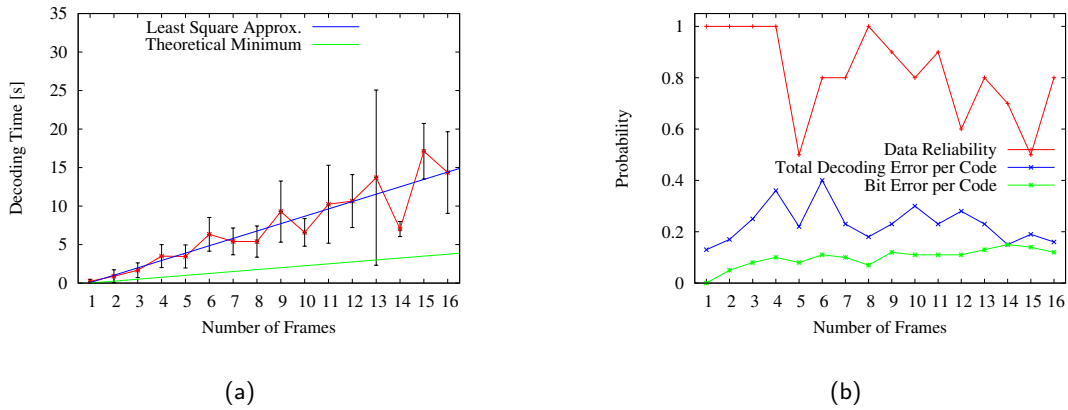


Figure 4.4.: Version 4: (a) Mean decoding times for sequences of 1 to 16 frames and their standard deviation. The theoretical minimum as well as a least square fit of the sample is also represented. (b) The data reliability, total decoding error and bit error for the corresponding sequence lengths.

There we can see that the reliability becomes an issue. Data reliability drop to 0.5 for some sequence length. The overall reliability mean is not better than 0.82. I would explain it as follows. First I assume that the probability that a code is recognized as a right code although it contains an error, which means there are 3 or more flipped bits in a code, is the same as for the version with 1 code per frame. But for that version 4 with 4 codes per frame, there are 4 times more codes in a sequence so the probability that a sequence contains a faulty code is about 4 times higher:

$$\begin{aligned}
 P(\text{error in the sequence version 4}) &= P(A_1 \cup \dots \cup A_4) \simeq P(A_1) + \dots + P(A_4) = \\
 &= 4P(\text{error in the sequence version 1})
 \end{aligned}$$

where $P(A_i)$ represents the probability that there is an error in the i th fourth of the sequence, according to the additivity rule of probability

$$P(U \cup V) = P(U) + P(V) - P(U \cap V)$$

if $P(U)$, $P(V)$ are small, as in our case, and U and V are independent, what we can assume, $P(U \cap V) = P(U)P(V) \simeq 0$ for set U , V satisfying the mentioned conditions.

For the version 1 with 1 code per frame, there are 4 out of 151 sequences that shows CRC checksum mismatch. Let's assume that this mismatch is caused by only one faulty code in the sequence. The estimated probability \hat{p}_1 that a sequence contains an error is then $\hat{p}_1 = 4/151 = 2.65\%$. We would then expect a sequence error probability for the version 4 of $p_4 \cong 4\hat{p}_1 = 10.6\%$. Unfortunately, the estimated error probability is with $\hat{p}_4 = 29/160 = 18.1\%$ much larger than that. There is however another difference in the setup between the two versions. For the version 4, each code needs to be much smaller than for the version 1, because 4 codes need to fit on the picture. A smaller code creates additional uncertainty in the decoding process and there is a size beyond which a code cannot be decoded anymore. I would then assume that this difference between p_4 and \hat{p}_4 can be partly explained because of this.

4.2.3. Throughput Comparison

Let's compare the two versions and have a look at the graph in figure 4.5. We see that the decoding time of the version 4 with 4 codes per frame seems a bit better than the version 1 with 1 code per frame. Until 10 frames or the label A on the graph, it can be interesting to use the version 1. For longer sequences it seems interesting to switch to the version 4, beginning with 3 frames. That version seems useful until 8 frames or the label B or about 270 bytes of data. For longer sequences, as we have seen before, the standard deviation grows too much, what can be annoying for a user if a worst case scenario happens.

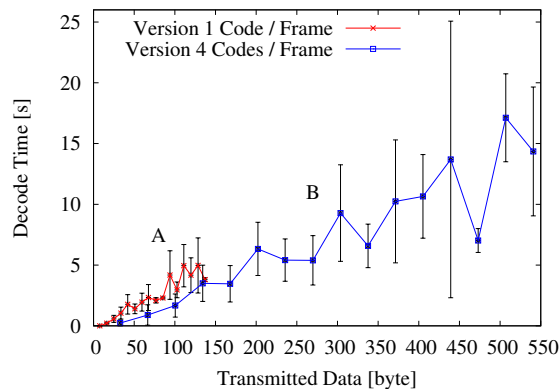


Figure 4.5.: Decoding time in function of the data length for the 2 versions

4.3. Frame Interval on a Faster Device

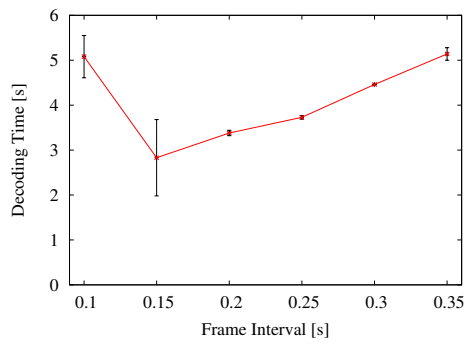
The idea of this third test is to look at the performance of a newer and faster device, the Nokia 6630 smartphone, equipped with a 220 MHz processor². The same optimization test as for the Nokia 6600 of section 4.1 has been performed, that means the measurements of the decoding time for different values of the frame interval Δt from 0.1 s to 0.35 s, 5 measurements per setup.

We can see in figure 4.6(a) the mean decoding time for that test along with the standard deviation. The best value is reached here for an interval time of 0.15 s, compared to 0.20 to 0.25 s for the Nokia 6600.

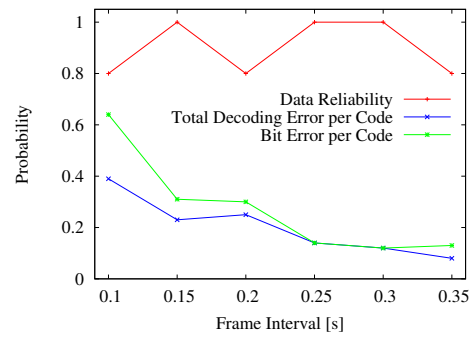
It gives an idea of the evolution in the throughput of the visual code sequence system. The best interval time, that has been used, for the Nokia 6600 was 0.25 s. With a value of 0.15 s, almost twice faster, the decoding time of a sequence is largely reduced. That gives the possibility to transmit more data than the 270 bytes like for a Nokia 6600.

That evolution of faster processors in smartphones is undoubtedly going to continue. That will enable to push the current limit in the capacity and usability of the visual code sequence system.

²Compared to a 104 MHz processor on the Nokia 6600.



(a)



(b)

Figure 4.6.: Optimization of the frame interval on a faster Nokia 6630. (a) Mean decoding time for a sequence of 16 frames, version 1, in function of the frame interval time Δt . The standard deviation is also represented. (b) The data reliability, total decoding error and bit error for the corresponding interval time Δt .

5. Related Work

Diverse papers have been published in the field of visual communication between mobile devices and device association.

The recognition of 2-dimensional barcodes using camera equipped smartphones have been described in numerous projects including Semacode [Sema], that enables to encode an URL, and Spotcode [Spot].

In the domain of device association Seeing-Is-Believing [McCu04] suggests to use a visual communication to perform a strong authentication between two devices. The authentication is done in each direction by displaying on the device screen a hash code of the device public key. The public key is then transmitted on the radio channel and verified using the hash code. An authentication in only one direction can be done using a visual code printed on a sticker located on the device. In that paper, a mention is also made about the possibility to use a set of codes instead of a single one if the data capacity is larger than what is supported by one code.

Madhavapeddy et al. [Sco05] describe how the Bluetooth address and a cookie¹ can be stored in a visual code in order for a mobile device to skip the Bluetooth Device Discovery phase and establish a direct connection to another device. In the example presented, a connection is made to a server through an HTTP GET request, the cookie providing an application id used to download a specific file.

¹A serial number.

6. Conclusion

The visual code sequence system represents an interesting way to transmit, as we have seen in the performance section, up to approximately 270 bytes of data¹ through the visual channel in a time acceptable by the user. It provides advantages in term of privacy compared to a communication entirely based on the radio channel—like wireless LAN and Bluetooth.

It enables to perform a communication initiation between two devices without having to disclose a physical address and is adapted for the preliminary transmission of meta-data.

As we have seen in the performance section, the data reliability of the system is in its present form not 100%. It would be probably advisable to improve that point. I see here two possible directions. On the one hand to integrate the CRC checksum in the code sequence system, what is in the present form done additionally at the application level. It would guarantee that an error in the transmission would be recognized and the corresponding signal sent to the client program. On the other hand, the coding scheme could be changed to enable the recognition and correction of more errors. A possibility would be the use of a Reed-Solomon error correcting code, as described in [McCu04]², enabling a payload of 68 bits and a redundancy of 15 bits.

The performances presented in this report are going to be gradually improved as the processing power of the smartphone devices will increase, as shown in section 4.3. Larger data capacity and shorter decoding time will result.

¹With processor speed of the 2.5G smartphone type like the Nokia 6600.

²On pages 15-16.

A. Using the Emulator for Debugging

A.1. General Consideration

The use of the emulator in the development process is very helpful. In particular because a bug happening on the target environment, the mobile phone, has often the effect of closing the application without any other information. The emulator enables to set breakpoints, to inspect the contents of the local variables, of the instantiated objects and of the memory. It is much faster to find a bug than using some logging strategies directly on the phone and having to do a lot of operations for each test, not to mention the repetitive installations of the application.

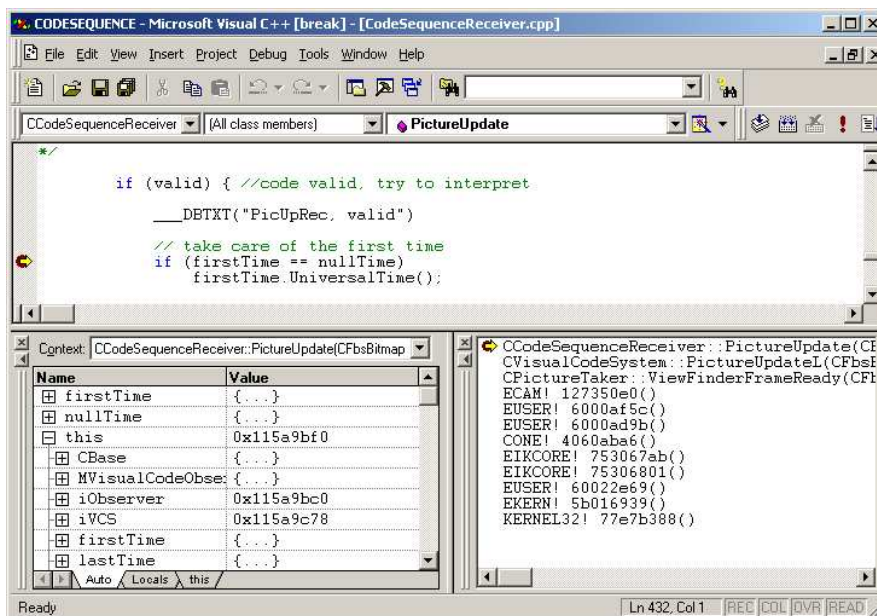


Figure A.1.: Debugger of Microsoft Visual Studio 6.0

The debugger from the IDE Microsoft Visual Studio 6.0 is started by pressing F5. The necessary classes are compiled on the spot. The first time a debugging session is launched, the location of the emulator executable for the `wins udeb` target has to be specified, it can be found at `epoc32/release/wins/udeb/epoc.exe` inside the `Nokia_6600_Camera_Plugin` or `Series60_v20` directory of the SDK tree, depending if the camera plugin SDK or the usual series 60 SDK is used.

A.2. Video Sequences as Camera Input

As the camera is used in the receiver module, it was necessary to find a way to use own video clips inside the emulator, which takes as video input a video file of YUV 422 format, called `handshake_<size>.yuv` and located in `7.0s/Nokia_6600_Camera_Plugin/epoc32/wins/c.`

<size> represents the video clip size and can be cif, qqvga, qvga sqcif or vga. In my case just the handshake.qqvga.yuv was actually needed.

Fortunately the webpage [Pulli] offers some documentation about that topic as well as a python script that converts a set of single JPEG images into a YUV 422 movie clip.



Figure A.2.: Epoc emulator. The default video sequence has been replaced by a custom sequence for testing.

It is first needed to convert a .avi movie clip—recorded for example using a webcam—into a set of JPEG images. That can be done under Linux using mplayer:

```
mplayer -vo jpeg -ao null <video_file.avi>
```

Then the python script of [Pulli] can be used:

```
python convert_yuv.py 0000
```

where 0000 is just the images prefix as created by mplayer. That's it. We can now use that movie clip instead of the original one.

Here is a simple bash script that gets the process automated:

```
#!/bin/bash
TMPDIR=tmp

# check param
if [ $# -ne 1 ]; then
    echo "usage: $0 <avi file>"
    exit 1;
fi
```

```
# create temp dir
mkdir $TMPDIR
cp $1 $TMPDIR/
cd $TMPDIR

# convert
mplayer -vo jpeg -ao null $1
python ../convert_yuv.py 000

# cleanup
mv handshake_qqvga.yuv ../${1/.avi/.yuv}
cd ..
rm -r $TMPDIR
```

The version of the tools used for the procedure is MPlayer 1.0pre5try2-3.3.4 and python 2.3.4-r1 as well as the imaging package for python, imaging 1.1.4-r1¹.

¹Under Gentoo Linux.

B. Raw Measurements

B.1. Frame Interval

Table B.1 Decoding time of a sequence of 16 frames for different frame intervals Δt between 0.10 and 0.35 s, 5 measurements per setup.

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]		Bit Error	Tot. Error	Length [bit]
209	0,1	1	1	16	17,19		0,54	0,58	1097
210	0,1	1	1	16	20,48		0,44	0,57	1097
211	0,1	1	1	16	12,59		0,41	0,48	1097
212	0,1	0	1	16	7,30		0,38	0,41	1097
213	0,1	0	1	16	11,16		0,37	0,39	1097
		\bar{x} 0,60			\bar{x} 16,76	σ 3,96	\bar{x} 0,43	\bar{x} 0,49	
214	0,15	1	1	16	5,80		0,17	0,23	1097
215	0,15	1	1	16	3,36		0,17	0,26	1097
216	0,15	1	1	16	9,33		0,20	0,45	1097
217	0,15	1	1	16	7,67		0,24	0,36	1097
218	0,15	1	1	16	11,08		0,19	0,27	1097
		\bar{x} 1,00			\bar{x} 7,45	σ 3,01	\bar{x} 0,19	\bar{x} 0,31	
233	0,2	1	1	16	3,11		0,08	0,08	1097
234	0,2	1	1	16	3,08		0,36	0,36	1097
235	0,2	1	1	16	3,06		0,00	0,04	1097
236	0,2	1	1	16	9,05		0,24	0,52	1097
237	0,2	1	1	16	8,13		0,08	0,53	1097
		\bar{x} 1,00			\bar{x} 5,28	σ 3,03	\bar{x} 0,15	\bar{x} 0,31	
238	0,25	1	1	16	3,64		0,05	0,33	1097
239	0,25	1	1	16	3,69		0,11	0,20	1097
240	0,25	1	1	16	3,84		0,17	0,17	1097
241	0,25	1	1	16	7,61		0,08	0,37	1097
242	0,25	1	1	16	8,78		0,09	0,39	1097
		\bar{x} 1,00			\bar{x} 5,51	σ 2,48	\bar{x} 0,10	\bar{x} 0,29	
243	0,3	1	1	16	4,45		0,07	0,24	1091
244	0,3	1	1	16	7,50		0,11	0,36	1091
245	0,3	1	1	16	7,55		0,13	0,36	1091
246	0,3	1	1	16	8,56		0,12	0,42	1091
247	0,3	1	1	16	5,45		0,15	0,24	1091
		\bar{x} 1,00			\bar{x} 6,70	σ 1,69	\bar{x} 0,11	\bar{x} 0,32	

Continued on next page

Table B.1 – continued from previous page

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]	Bit Error	Tot. Error	Length [bit]
307	0,35	1	1	16	6,20	0,03	0,26	1096
308	0,35	1	1	16	5,28	0,06	0,21	1096
309	0,35	1	1	16	9,92	0,16	0,35	1096
310	0,35	1	1	16	9,48	0,16	0,37	1096
311	0,35	1	1	16	9,69	0,14	0,36	1096
		\bar{x} 1,00			\bar{x} σ 8,12 2,20	\bar{x}	\bar{x} 0,11 0,31	

Table B.2 Same as before for the 2 frame intervals Δt of 0.20 and 0.25 s, 20 measurements per setup.

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]	Bit Error	Tot. Error	Length [bit]
312	0,2	1	1	16	3,05	0,05	0,31	1096
313	0,2	1	1	16	7,00	0,31	0,33	1096
314	0,2	1	1	16	12,19	0,22	0,39	1096
315	0,2	1	1	16	5,98	0,35	0,35	1096
316	0,2	1	1	16	3,16	0,22	0,22	1096
317	0,2	1	1	16	8,28	0,16	0,22	1096
318	0,2	1	1	16	8,53	0,18	0,35	1096
319	0,2	1	1	16	4,73	0,29	0,29	1096
320	0,2	1	1	16	8,64	0,31	0,44	1096
321	0,2	1	1	16	9,55	0,34	0,44	1096
322	0,2	1	1	16	9,25	0,24	0,41	1096
323	0,2	1	1	16	8,00	0,22	0,38	1096
324	0,2	1	1	16	6,77	0,00	0,34	1096
325	0,2	1	1	16	14,20	0,29	0,44	1096
326	0,2	1	1	16	6,13	0,32	0,40	1096
327	0,2	1	1	16	23,59	0,21	0,39	1096
328	0,2	1	1	16	10,39	0,27	0,39	1096
329	0,2	1	1	16	8,20	0,08	0,25	1096
331	0,2	1	1	16	6,67	0,28	0,43	1096
332	0,2	1	1	16	6,34	0,22	0,40	1096
		\bar{x} 1,00			\bar{x} σ 8,53 4,46	\bar{x}	\bar{x} 0,23 0,36	
333	0,25	1	1	16	6,95	0,19	0,23	1096
334	0,25	1	1	16	11,28	0,13	0,45	1096
335	0,25	1	1	16	6,89	0,18	0,44	1096
336	0,25	1	1	16	7,45	0,15	0,45	1096
337	0,25	1	1	16	15,69	0,12	0,46	1096
338	0,25	1	1	16	29,42	0,14	0,50	1096
339	0,25	1	1	16	7,39	0,18	0,38	1096
340	0,25	1	1	16	7,56	0,15	0,35	1096
341	0,25	1	1	16	4,58	0,18	0,26	1096
342	0,25	1	1	16	7,63	0,14	0,27	1096

Continued on next page

Table B.2 – continued from previous page

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]	Bit Error	Tot. Error	Length [bit]
343	0,25	0	1	16	7,34	0,16	0,37	1096
344	0,25	1	1	16	6,50	0,08	0,18	1096
345	0,25	1	1	16	6,78	0,13	0,27	1096
346	0,25	1	1	16	3,72	0,12	0,12	1096
347	0,25	1	1	16	3,75	0,17	0,20	1096
348	0,25	1	1	16	3,77	0,20	0,20	1096
349	0,25	1	1	16	3,78	0,12	0,12	1096
350	0,25	1	1	16	3,78	0,08	0,12	1096
351	0,25	1	1	16	3,77	0,08	0,08	1096
352	0,25	1	1	16	3,67	0,13	0,13	1096
		\bar{x} 0,95			\bar{x} σ 7,59 5,94	\bar{x}	\bar{x}	

B.2. Sequence Decoding Time

Table B.3 Decoding time of sequences of 1 to 16 frames of version 1, 1 visual code per frame, 10 measurements per setup.

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]	Bit Error	Tot. Error	Length [bit]
379	0,25	1	1	1	0,00	0,00	0,00	61
		\bar{x} 1			\bar{x} σ 0,00 0,00	\bar{x}	\bar{x}	[byte] 7,625
618	0,25	1	1	2	0,13	0,00	0,00	131
619	0,25	1	1	2	0,27	0,00	0,00	131
620	0,25	1	1	2	0,13	0,00	0,00	131
621	0,25	1	1	2	0,13	0,00	0,00	131
622	0,25	1	1	2	0,27	0,00	0,00	131
623	0,25	1	1	2	0,25	0,00	0,33	131
624	0,25	1	1	2	0,27	0,00	0,25	131
625	0,25	1	1	2	0,27	0,00	0,00	131
626	0,25	1	1	2	0,25	0,00	0,33	131
627	0,25	1	1	2	0,27	0,00	0,00	131
		\bar{x} 1			\bar{x} σ 0,22 0,07	\bar{x}	\bar{x}	[byte] 16,375
380	0,25	1	1	3	0,55	0,00	0,20	200
381	0,25	1	1	3	0,52	0,00	0,40	200
382	0,25	1	1	3	0,47	0,40	0,25	200
383	0,25	1	1	3	0,56	0,00	0,20	200
384	0,25	1	1	3	0,45	0,25	0,25	200
385	0,25	1	1	3	0,45	0,25	0,40	200
386	0,25	1	1	3	0,44	0,00	0,57	200
387	0,25	1	1	3	1,39	0,20	0,67	200
388	0,25	1	1	3	0,44	0,25	0,25	200

Continued on next page

Table B.3 – continued from previous page

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]		Bit Error	Tot. Error	Length [bit]
389	0,25	1	1	3	0,44		0,25	0,40	200
		\bar{x} 1			\bar{x} 0,57	σ 0,29	\bar{x} 0,16	\bar{x} 0,36	[byte] 25
628	0,25	1	1	4	1,44		0,00	0,43	269
629	0,25	1	1	4	0,55		0,20	0,33	269
630	0,25	1	1	4	1,70		0,30	0,53	269
631	0,25	1	1	4	0,70		0,14	0,14	269
632	0,25	1	1	4	0,72		0,00	0,00	269
633	0,25	1	1	4	1,75		0,13	0,53	269
634	0,25	1	1	4	0,69		0,17	0,17	269
635	0,25	1	1	4	0,70		0,17	0,17	269
636	0,25	1	1	4	0,69		0,00	0,00	269
637	0,25	1	1	4	1,53		0,29	0,64	269
		\bar{x} 1			\bar{x} 1,05	σ 0,49	\bar{x} 0,14	\bar{x} 0,29	[byte] 33,625
395	0,25	1	1	5	1,98		0,11	0,50	336
396	0,25	1	1	5	2,03		0,25	0,40	336
397	0,25	1	1	5	0,88		0,14	0,14	336
398	0,25	1	1	5	2,28		0,07	0,22	336
399	0,25	1	1	5	3,27		0,05	0,25	336
400	0,25	1	1	5	2,28		0,10	0,53	336
401	0,25	1	1	5	0,88		0,14	0,14	336
402	0,25	1	1	5	1,02		0,00	0,00	336
403	0,25	1	1	5	2,06		0,17	0,38	336
404	0,25	1	1	5	1,00		0,13	0,13	336
		\bar{x} 1			\bar{x} 1,77	σ 0,80	\bar{x} 0,12	\bar{x} 0,27	[byte] 42
639	0,25	0	1	6	3,72		0,05	0,31	407
640	0,25	1	1	6	1,27		0,13	0,36	407
641	0,25	1	1	6	1,23		0,09	0,09	407
642	0,25	1	1	6	1,20		0,22	0,36	407
643	0,25	1	1	6	2,34		0,07	0,38	407
644	0,25	1	1	6	1,77		0,25	0,40	407
645	0,25	1	1	6	1,11		0,30	0,46	407
646	0,25	1	1	6	1,14		0,33	0,45	407
647	0,25	1	1	6	1,30		0,27	0,27	407
648	0,25	1	1	6	1,27		0,10	0,10	407
		\bar{x} 0,9			\bar{x} 1,40	σ 0,40	\bar{x} 0,18	\bar{x} 0,32	[byte] 50,875
405	0,25	1	1	7	1,41		0,00	0,00	476
406	0,25	1	1	7	1,58		0,10	0,36	476
407	0,25	1	1	7	1,53		0,18	0,25	476
408	0,25	1	1	7	1,42		0,09	0,09	476
409	0,25	1	1	7	2,78		0,27	0,50	476

Continued on next page

Table B.3 – continued from previous page

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]	Bit Error	Tot. Error	Length [bit]
410	0,25	1	1	7	3,16	0,25	0,42	476
411	0,25	1	1	7	3,08	0,31	0,56	476
412	0,25	1	1	7	1,55	0,08	0,08	476
413	0,25	1	1	7	1,55	0,25	0,25	476
414	0,25	1	1	7	1,41	0,27	0,27	476
		\bar{x} 1			\bar{x} σ 1,95 0,74	\bar{x} \bar{x} 0,18 0,28		[byte] 59,5
649	0,25	1	1	8	4,78	0,25	0,61	540
650	0,25	1	1	8	3,72	0,15	0,45	540
651	0,25	1	1	8	1,78	0,14	0,20	540
652	0,25	1	1	8	1,78	0,21	0,35	540
653	0,25	1	1	8	1,78	0,31	0,64	540
654	0,25	1	1	8	1,73	0,17	0,29	540
655	0,25	1	1	8	1,67	0,15	0,15	540
656	0,25	1	1	8	1,94	0,14	0,20	540
657	0,25	1	1	8	2,45	0,06	0,16	540
658	0,25	1	1	8	1,67	0,00	0,08	540
		\bar{x} 1			\bar{x} σ 2,33 1,07	\bar{x} \bar{x} 0,16 0,31		[byte] 67,5
415	0,25	1	1	9	2,44	0,19	0,41	614
416	0,25	1	1	9	2,55	0,26	0,30	614
417	0,25	1	1	9	2,11	0,12	0,12	614
418	0,25	1	1	9	2,09	0,00	0,00	614
419	0,25	1	1	9	1,95	0,19	0,19	614
420	0,25	1	1	9	1,92	0,07	0,07	614
421	0,25	1	1	9	1,88	0,08	0,33	614
422	0,25	1	1	9	1,95	0,13	0,13	614
423	0,25	1	1	9	2,05	0,12	0,12	614
424	0,25	1	1	9	2,03	0,13	0,19	614
		\bar{x} 1			\bar{x} σ 2,10 0,22	\bar{x} \bar{x} 0,13 0,18		[byte] 76,75
659	0,25	1	1	10	2,34	0,06	0,17	682
660	0,25	1	1	10	2,31	0,00	0,00	682
661	0,25	1	1	10	2,42	0,28	0,28	682
662	0,25	1	1	10	2,17	0,00	0,00	682
663	0,25	0	1	10	0,73	0,29	0,17	239
664	0,25	0	1	10	2,31	0,06	0,00	682
665	0,25	1	1	10	2,31	0,24	0,24	682
666	0,25	1	1	10	2,28	0,18	0,18	682
667	0,25	1	1	10	2,17	0,19	0,19	682
668	0,25	1	1	10	2,30	0,12	0,12	682
		\bar{x} 0,8			\bar{x} σ 2,29 0,08	\bar{x} \bar{x} 0,14 0,13		[byte] 85,25
425	0,25	1	1	11	2,58	0,12	0,21	752

Continued on next page

Table B.3 – continued from previous page

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]	Bit Error	Tot. Error	Length [bit]	
426	0,25	1	1	11	4,47	0,15	0,30	752	
427	0,25	1	1	11	3,02	0,32	0,43	752	
428	0,25	1	1	11	2,59	0,17	0,21	752	
429	0,25	1	1	11	7,25	0,13	0,40	752	
430	0,25	1	1	11	4,09	0,08	0,33	752	
431	0,25	1	1	11	7,91	0,20	0,42	752	
432	0,25	1	1	11	4,89	0,09	0,23	752	
433	0,25	1	1	11	2,47	0,05	0,05	752	
434	0,25	1	1	11	2,48	0,16	0,16	752	
		\bar{x} 1			\bar{x} 4,18	σ 2,00	\bar{x} 0,15	\bar{x} 0,28	[byte] 94
669	0,25	1	1	12	4,80	0,16	0,26	821	
670	0,25	1	1	12	2,67	0,16	0,20	821	
671	0,25	1	1	12	2,83	0,14	0,14	821	
672	0,25	1	1	12	2,72	0,10	0,10	821	
673	0,25	1	1	12	2,84	0,25	0,29	821	
674	0,25	1	1	12	2,70	0,15	0,15	821	
675	0,25	1	1	12	2,73	0,05	0,05	821	
676	0,25	1	1	12	2,86	0,24	0,24	821	
677	0,25	1	1	12	2,69	0,25	0,25	821	
678	0,25	1	1	12	2,78	0,20	0,20	821	
		\bar{x} 1			\bar{x} 2,96	σ 0,65	\bar{x} 0,17	\bar{x} 0,19	[byte] 102,625
435	0,25	1	1	13	5,98	0,09	0,38	888	
436	0,25	1	1	13	5,81	0,21	0,30	888	
437	0,25	1	1	13	4,17	0,15	0,37	888	
438	0,25	1	1	13	7,84	0,19	0,55	888	
439	0,25	1	1	13	3,06	0,09	0,19	888	
440	0,25	0	1	13	3,11	0,04	0,08	888	
441	0,25	1	1	13	3,03	0,09	0,13	888	
442	0,25	1	1	13	4,81	0,21	0,32	888	
443	0,25	1	1	13	3,16	0,21	0,24	888	
444	0,25	1	1	13	6,67	0,21	0,35	888	
		\bar{x} 0,9			\bar{x} 4,95	σ 1,74	\bar{x} 0,15	\bar{x} 0,29	[byte] 111
679	0,25	1	1	14	3,13	0,00	0,09	958	
680	0,25	1	1	14	4,75	0,00	0,14	958	
681	0,25	1	1	14	4,11	0,13	0,41	958	
682	0,25	1	1	14	3,33	0,17	0,21	958	
683	0,25	1	1	14	3,27	0,26	0,29	958	
684	0,25	1	1	14	3,30	0,00	0,04	958	
685	0,25	1	1	14	6,61	0,17	0,33	958	
686	0,25	1	1	14	3,11	0,24	0,36	958	
687	0,25	1	1	14	6,80	0,23	0,25	958	

Continued on next page

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]		Bit Error	Tot. Error	Length [bit]
688	0,25	1	1	14	3,31		0,13	0,13	958
		\bar{x} 1			\bar{x} 4,17	σ 1,43	\bar{x} 0,13	\bar{x} 0,23	[byte] 119,75
445	0,25	1	1	15	3,61		0,00	0,33	1028
446	0,25	1	1	15	4,77		0,14	0,26	1028
447	0,25	1	1	15	7,25		0,03	0,36	1028
448	0,25	1	1	15	10,11		0,04	0,32	1028
449	0,25	1	1	15	3,52		0,12	0,12	1028
450	0,25	1	1	15	3,48		0,24	0,24	1028
451	0,25	1	1	15	3,44		0,08	0,04	1028
452	0,25	1	1	15	3,61		0,19	0,19	1028
453	0,25	1	1	15	3,45		0,20	0,20	1028
455	0,25	1	1	15	3,61		0,04	0,04	1028
456	0,25	1	1	15	6,44		0,17	0,30	1028
		\bar{x} 1			\bar{x} 4,97	σ 2,27	\bar{x} 0,12	\bar{x} 0,21	[byte] 128,5
689	0,25	1	1	16	4,19		0,07	0,13	1097
690	0,25	1	1	16	3,73		0,07	0,07	1097
691	0,25	1	1	16	3,91		0,18	0,18	1097
692	0,25	1	1	16	3,73		0,00	0,00	1097
693	0,25	1	1	16	3,73		0,07	0,07	1097
694	0,25	1	1	16	3,81		0,00	0,00	1097
695	0,25	1	1	16	3,72		0,04	0,04	1097
696	0,25	1	1	16	3,72		0,26	0,26	1097
697	0,25	1	1	16	3,73		0,00	0,00	1097
698	0,25	1	1	16	3,69		0,07	0,07	1097
		\bar{x} 1			\bar{x} 3,80	σ 0,15	\bar{x} 0,08	\bar{x} 0,08	[byte] 137,125

Table B.4 Same as before for the version 4, 4 visual codes per frame.

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]		Bit Error	Tot. Error	Length [bit]
458	0,25	1	4	1	0,00		0,00	0,00	262
459	0,25	1	4	1	0,39		0,00	0,33	262
460	0,25	1	4	1	0,22		0,00	0,13	262
461	0,25	1	4	1	0,22		0,00	0,13	262
462	0,25	1	4	1	0,38		0,00	0,33	262
463	0,25	1	4	1	0,00		0,00	0,00	262
464	0,25	1	4	1	0,83		0,00	0,25	262
465	0,25	1	4	1	0,23		0,00	0,13	262
466	0,25	1	4	1	0,00		0,00	0,00	262
467	0,25	1	4	1	0,00		0,00	0,00	262
		\bar{x} 1			\bar{x} 0,23	σ 0,26	\bar{x} 0,00	\bar{x} 0,13	[byte] 32,75

Continued on next page

Table B.4 – continued from previous page

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]	Bit Error	Tot. Error	Length [bit]	
468	0,25	1	4	2	3,05	0,13	0,34	533	
469	0,25	1	4	2	0,23	0,00	0,00	533	
470	0,25	1	4	2	0,80	0,13	0,30	533	
471	0,25	1	4	2	0,44	0,00	0,08	533	
472	0,25	1	4	2	0,23	0,00	0,00	533	
473	0,25	1	4	2	0,44	0,09	0,17	533	
474	0,25	1	4	2	0,86	0,00	0,10	533	
475	0,25	1	4	2	0,77	0,00	0,25	533	
476	0,25	1	4	2	0,63	0,08	0,25	533	
477	0,25	1	4	2	1,50	0,07	0,22	533	
		\bar{x}			\bar{x}	σ	\bar{x}	\bar{x}	[byte]
		1			0,89	0,84	0,05	0,17	66,625
478	0,25	1	4	3	1,66	0,13	0,28	804	
479	0,25	1	4	3	1,22	0,14	0,32	804	
480	0,25	1	4	3	2,13	0,03	0,35	804	
481	0,25	1	4	3	2,02	0,09	0,30	804	
482	0,25	1	4	3	3,91	0,13	0,43	804	
483	0,25	1	4	3	1,28	0,08	0,21	804	
484	0,25	1	4	3	0,44	0,00	0,00	804	
485	0,25	1	4	3	0,89	0,05	0,10	804	
486	0,25	1	4	3	1,97	0,19	0,25	804	
487	0,25	1	4	3	1,20	0,00	0,25	804	
		\bar{x}			\bar{x}	σ	\bar{x}	\bar{x}	[byte]
		1			1,67	0,95	0,08	0,25	100,5
488	0,25	1	4	4	5,33	0,20	0,46	1074	
489	0,25	1	4	4	5,05	0,20	0,53	1074	
490	0,25	1	4	4	1,67	0,03	0,22	1074	
491	0,25	1	4	4	4,88	0,04	0,52	1074	
492	0,25	1	4	4	0,69	0,00	0,00	1074	
493	0,25	1	4	4	3,64	0,11	0,37	1074	
494	0,25	1	4	4	4,33	0,07	0,43	1074	
495	0,25	1	4	4	3,13	0,02	0,30	1074	
496	0,25	1	4	4	3,09	0,15	0,39	1074	
497	0,25	1	4	4	3,22	0,20	0,36	1074	
		\bar{x}			\bar{x}	σ	\bar{x}	\bar{x}	[byte]
		1			3,50	1,49	0,10	0,36	134,25
498	0,25	1	4	5	4,48	0,12	0,34	1344	
499	0,25	0	4	5	2,53	0,12	0,27	1344	
500	0,25	1	4	5	3,17	0,09	0,15	1344	
501	0,25	0	4	5	1,11	0,04	0,08	1344	
502	0,25	1	4	5	1,16	0,00	0,04	1344	
503	0,25	0	4	5	2,47	0,05	0,29	1344	
504	0,25	0	4	5	5,05	0,17	0,35	1344	
505	0,25	0	4	5	3,02	0,12	0,25	1344	

Continued on next page

Table B.4 – continued from previous page

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]	Bit Error	Tot. Error	Length [bit]	
506	0,25	1	4	5	5,08	0,07	0,24	1344	
507	0,25	1	4	5	3,42	0,02	0,16	1344	
		\bar{x} 0,5			\bar{x} 3,46	σ 1,50	\bar{x} 0,08	\bar{x} 0,22	[byte] 168
508	0,25	1	4	6	2,89	0,08	0,42	1617	
509	0,25	1	4	6	6,25	0,10	0,38	1617	
510	0,25	1	4	6	5,58	0,14	0,41	1617	
511	0,25	1	4	6	6,08	0,09	0,42	1617	
512	0,25	1	4	6	10,00	0,14	0,46	1617	
513	0,25	1	4	6	5,19	0,06	0,33	1617	
514	0,25	1	4	6	5,84	0,07	0,40	1617	
515	0,25	0	4	6	8,64	0,09	0,38	1617	
516	0,25	1	4	6	8,80	0,10	0,43	1617	
517	0,25	0	4	6	4,17	0,20	0,39	1617	
		\bar{x} 0,8			\bar{x} 6,33	σ 2,19	\bar{x} 0,11	\bar{x} 0,40	[byte] 202,125
518	0,25	1	4	7	3,44	0,13	0,19	1885	
519	0,25	1	4	7	3,36	0,04	0,16	1885	
520	0,25	1	4	7	4,56	0,11	0,24	1885	
521	0,25	0	4	7	6,89	0,12	0,28	1885	
522	0,25	1	4	7	6,14	0,13	0,31	1885	
523	0,25	1	4	7	5,36	0,09	0,16	1885	
524	0,25	1	4	7	8,70	0,13	0,24	1885	
525	0,25	1	4	7	5,30	0,09	0,26	1885	
526	0,25	0	4	7	4,19	0,12	0,30	1885	
527	0,25	1	4	7	6,41	0,06	0,17	1885	
		\bar{x} 0,8			\bar{x} 5,41	σ 1,74	\bar{x} 0,10	\bar{x} 0,23	[byte] 235,625
528	0,25	1	4	8	4,80	0,08	0,21	2158	
529	0,25	1	4	8	2,89	0,06	0,16	2158	
530	0,25	1	4	8	4,08	0,09	0,16	2158	
531	0,25	1	4	8	6,13	0,13	0,20	2158	
532	0,25	1	4	8	4,61	0,06	0,18	2158	
533	0,25	1	4	8	5,91	0,07	0,14	2158	
534	0,25	1	4	8	9,36	0,10	0,25	2158	
535	0,25	1	4	8	7,27	0,12	0,30	2158	
536	0,25	1	4	8	6,22	0,05	0,14	2158	
537	0,25	1	4	8	2,67	0,00	0,10	2158	
		\bar{x} 1			\bar{x} 5,39	σ 2,03	\bar{x} 0,07	\bar{x} 0,18	[byte] 269,75
538	0,25	1	4	9	6,91	0,15	0,22	2430	
539	0,25	1	4	9	15,48	0,11	0,26	2430	
540	0,25	1	4	9	4,39	0,09	0,13	2430	
541	0,25	0	4	9	8,05	0,15	0,19	2430	

Continued on next page

Table B.4 – continued from previous page

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]	Bit Error	Tot. Error	Length [bit]	
542	0,25	1	4	9	7,42	0,07	0,24	2430	
543	0,25	1	4	9	14,08	0,12	0,21	2430	
544	0,25	1	4	9	5,63	0,14	0,24	2430	
545	0,25	1	4	9	6,80	0,12	0,21	2430	
546	0,25	1	4	9	10,30	0,19	0,32	2430	
547	0,25	1	4	9	12,56	0,09	0,25	2430	
		\bar{x} 0,9			\bar{x} 9,28	σ 3,97	\bar{x} 0,12	\bar{x} 0,23	[byte] 303,75
548	0,25	1	4	10	6,27	0,09	0,20	2701	
549	0,25	1	4	10	4,19	0,14	0,25	2701	
550	0,25	1	4	10	4,92	0,11	0,20	2701	
551	0,25	0	4	10	5,05	0,10	0,15	2701	
552	0,25	1	4	10	5,22	0,15	0,24	2701	
553	0,25	1	4	10	9,64	0,11	0,30	2701	
554	0,25	1	4	10	7,09	0,13	0,36	2701	
555	0,25	0	4	10	14,31	0,10	0,50	2701	
556	0,25	1	4	10	7,88	0,08	0,37	2701	
557	0,25	1	4	10	7,45	0,13	0,41	2701	
		\bar{x} 0,8			\bar{x} 6,58	σ 1,79	\bar{x} 0,11	\bar{x} 0,30	[byte] 337,625
558	0,25	1	4	11	21,20	0,14	0,23	2971	
559	0,25	1	4	11	6,75	0,10	0,27	2971	
560	0,25	1	4	11	10,52	0,09	0,25	2971	
561	0,25	1	4	11	5,78	0,14	0,18	2971	
562	0,25	0	4	11	7,13	0,11	0,23	2971	
563	0,25	1	4	11	9,94	0,08	0,19	2971	
564	0,25	1	4	11	9,41	0,12	0,27	2971	
565	0,25	1	4	11	15,02	0,13	0,26	2971	
566	0,25	1	4	11	8,39	0,10	0,23	2971	
567	0,25	1	4	11	5,19	0,08	0,23	2971	
		\bar{x} 0,9			\bar{x} 10,24	σ 5,06	\bar{x} 0,11	\bar{x} 0,23	[byte] 371,375
568	0,25	1	4	12	11,41	0,13	0,30	3241	
569	0,25	1	4	12	15,48	0,12	0,34	3241	
570	0,25	1	4	12	8,09	0,08	0,26	3241	
571	0,25	0	4	12	5,61	0,17	0,34	884	
572	0,25	0	4	12	9,50	0,13	0,32	1225	
573	0,25	1	4	12	11,16	0,13	0,21	3241	
574	0,25	0	4	12	3,48	0,09	0,29	884	
575	0,25	1	4	12	12,20	0,14	0,36	3241	
576	0,25	0	4	12	5,77	0,09	0,24	882	
577	0,25	1	4	12	5,55	0,07	0,16	3241	
		\bar{x} 0,6			\bar{x} 10,65	σ 3,44	\bar{x} 0,11	\bar{x} 0,28	[byte] 405,125

Continued on next page

Table B.4 – continued from previous page

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]	Bit Error	Tot. Error	Length [bit]
578	0,25	0	4	13	7,45	0,14	0,16	3514
579	0,25	1	4	13	12,89	0,15	0,22	3514
580	0,25	1	4	13	40,94	0,13	0,41	3514
581	0,25	0	4	13	13,70	0,11	0,45	3514
582	0,25	1	4	13	12,61	0,13	0,30	3514
583	0,25	1	4	13	10,20	0,07	0,30	3514
584	0,25	1	4	13	11,72	0,16	0,19	3514
585	0,25	1	4	13	5,13	0,16	0,10	3514
586	0,25	1	4	13	10,05	0,14	0,11	3514
587	0,25	1	4	13	5,98	0,10	0,10	3514
		\bar{x} 0,8			\bar{x} σ 13,69 11,38	\bar{x} \bar{x} 0,13 0,23		[byte] 439,25
588	0,25	1	4	14	7,05	0,09	0,20	3784
589	0,25	0	4	14	4,83	0,17	0,14	3784
590	0,25	1	4	14	5,80	0,23	0,19	3784
591	0,25	1	4	14	8,95	0,16	0,14	3784
592	0,25	1	4	14	6,75	0,16	0,15	3784
593	0,25	1	4	14	7,22	0,15	0,14	3784
594	0,25	1	4	14	6,42	0,12	0,11	3784
595	0,25	0	4	14	10,88	0,15	0,11	1844
596	0,25	0	4	14	6,13	0,11	0,12	3784
597	0,25	1	4	14	6,94	0,17	0,17	3784
		\bar{x} 0,7			\bar{x} σ 7,02 0,98	\bar{x} \bar{x} 0,15 0,15		[byte] 473
598	0,25	0	4	15	7,86	0,19	0,23	3913
599	0,25	0	4	15	14,67	0,18	0,17	4056
600	0,25	1	4	15	23,25	0,16	0,24	4056
601	0,25	1	4	15	15,83	0,14	0,22	4056
602	0,25	1	4	15	14,70	0,14	0,20	4056
603	0,25	1	4	15	14,44	0,13	0,19	4056
604	0,25	0	4	15	11,27	0,13	0,13	4056
605	0,25	1	4	15	17,38	0,10	0,17	4056
606	0,25	0	4	15	15,63	0,11	0,18	4056
607	0,25	0	4	15	9,19	0,15	0,21	4056
		\bar{x} 0,5			\bar{x} σ 17,12 3,62	\bar{x} \bar{x} 0,14 0,19		[byte] 507
608	0,25	1	4	16	19,52	0,10	0,17	4327
609	0,25	1	4	16	16,42	0,14	0,18	4327
610	0,25	1	4	16	8,27	0,11	0,13	4327
611	0,25	1	4	16	12,78	0,13	0,19	4327
612	0,25	1	4	16	7,95	0,07	0,11	4327
613	0,25	0	4	16	16,02	0,08	0,13	4327
614	0,25	0	4	16	20,09	0,15	0,17	4327
615	0,25	1	4	16	11,56	0,15	0,19	4327

Continued on next page

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]	Bit Error	Tot. Error	Length [bit]
616	0,25	1	4	16	23,11	0,11	0,15	4327
617	0,25	1	4	16	15,22	0,15	0,22	4327
		\bar{x} 0,8			\bar{x} σ 14,35 5,30	\bar{x} 0,12	\bar{x} 0,16	[byte] 540,875

B.3. Frame Interval on a Faster Device

Table B.5 Decoding time of a sequence of 16 frames for different frame intervals Δt on a faster smartphone device, the Nokia 6630.

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]	Bit Error	Tot. Error	Length [bit]
1	0,10	0	1	16	2,0625	0,35	0,89	1098
2	0,10	1	1	16	1,453125	0,33	0,58	1098
3	0,10	1	1	16	2,734375	0,47	0,47	1097
4	0,10	1	1	16	2,359375	0,41	0,55	1098
5	0,10	1	1	16	2,125	0,41	0,71	1097
		\bar{x} 0,80			\bar{x} σ 2,146875 0,47	\bar{x} 0,39	\bar{x} 0,64	
6	0,15	1	1	16	2,171875	0,29	0,32	1098
7	0,15	1	1	16	2,296875	0,23	0,25	1097
8	0,15	1	1	16	2,203125	0,24	0,24	1097
9	0,15	1	1	16	4,109375	0,25	0,25	1095
10	0,15	1	1	16	2,2046875	0,15	0,46	1097
		\bar{x} 1,00		2,5971875	\bar{x} σ 0,85 0,23	\bar{x} 0,31	\bar{x}	
11	0,20	1	1	16	3	0,36	0,37	1098
12	0,20	1	1	16	2,921875	0,02	0,05	1095
13	0,20	0	1	16	3,09375	0,24	0,36	1098
14	0,20	1	1	16	2,984375	0,28	0,27	1098
15	0,20	1	1	16	3	0,32	0,48	1097
		\bar{x} 0,80			\bar{x} σ 3 0,06	\bar{x} 0,25	\bar{x} 0,30	
16	0,25	1	1	16	3,65625	0,20	0,21	1098
17	0,25	1	1	16	3,734375	0,12	0,13	1098
18	0,25	1	1	16	3,71875	0,11	0,11	1097
19	0,25	1	1	16	3,6875	0,13	0,13	1098
20	0,25	1	1	16	3,75	0,14	0,14	1097
		\bar{x} 1,00			\bar{x} σ 3,709375 0,04	\bar{x} 0,14	\bar{x} 0,14	
21	0,30	1	1	16	4,4375	0,10	0,10	1097
22	0,30	1	1	16	4,453125	0,13	0,15	1097
23	0,30	1	1	16	4,453125	0,10	0,10	1098
24	0,30	1	1	16	4,4375	0,13	0,13	1097

Continued on next page

Table B.5 – continued from previous page

Measure	Δt [s]	CRC	Version	Nb Frames	Decoding Time [s]	Bit Error	Tot. Error	Length [bit]
25	0,30	1	1	16	4,453125	0,12	0,12	1092
		\bar{x} 1,00			\bar{x} σ 4,446875 0,01	\bar{x} 0,12	\bar{x} 0,12	
1	0,35	1	1	16	4,96875	0,09	0,20	1098
2	0,35	1	1	16	5,28125	0,09	0,13	1096
3	0,35	1	1	16	5,203125	0,10	0,13	1097
4	0,35	1	1	16	5,234375	0,07	0,08	1097
5	0,35	0	1	16	5	0,07	0,10	1098
		\bar{x} 0,80			\bar{x} σ 5,1375 0,14	\bar{x} 0,08	\bar{x} 0,13	

C. Hardware and Software Details

C.1. Smartphone

The standard device that has been used, in particular of the measurement part, is a Nokia 6600 with the following specifications:

Device	Nokia 6600
CPU	32-bit ARM-9, 104 MHz
Firmware	V 3.49.1
Flash Memory	6 MB + 32 MB on MMC card
RAM	10 MB
OS	Symbian OS 7.0s

The device used for comparison in section 4.3, of newer generation, is a Nokia 6630 that has the following specifications:

Device	Nokia 6630
CPU	32-bit ARM5, 220 MHz
Firmware	V 2.39.15
Flash Memory	10 MB + 64 MB on MMC card
RAM	unspecified
OS	Symbian OS 8.0a

C.2. Software

The following software has been used in the frame of the project:

OS	Windows 2000 and XP Professional Gentoo Linux with kernel 2.6
IDE	Microsoft Visual C++ 6.0 Eclipse 3.0.1
SDK	Symbian Series 60 SDK 2.0 Nokia 6600 Camera Plugin for Series 60 SDK 2.0 Sun J2SE 1.4.x SDK

Bibliography

- [Roh] Michael Rohs. *Visual Code System*.
<http://www.inf.ethz.ch/personal/rohs/visualcodes/>
- [Roh05] Michael Rohs and Philipp Zweifel, *A Conceptual Framework for Camera Phone-based Interaction Techniques*, May 2005
- [McCu04] Jonathan M. McCune, Adrian Perrig and Michael K. Reiter *Seeing-Is-Believing: Using Camera Phones for Human-Verifiable Authentication*, November 2004
- [SymC++] *Symbian OS: From ANSI C/C++ To Symbian C++*, Version 1.0, March 22, 2004.
http://www.forum.nokia.com/main/0,,21_30,00.html
- [Pulli] Kari Pulli. *Tool for using your own video clips in the Series 60 emulator*
<http://graphics.csail.mit.edu/~kapu/symbian.html>
- [Sco05] David Scott, Richard Sharp et al. *Using Visual Tags to Bypass Bluetooth Device Discovery*, January 2005
- [SDP] Eugene A. Gryazin. *Service Discovery in Bluetooth*
- [Rek97] Jun Rekimoto *Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments*, October 1997
- [Mad04] Anil Madhavapeddy, David Scott, Richard Sharp and Eben Upton. *Using Camera-Phones to Enhance Human-Computer Interaction*, September 2004
- [Sema] Semacode Website. <http://semacode.org/>
- [Spot] High Energy Magic Ltd. Spotcode. <http://www.highenergymagic.com/>
- [NewLC] NewLC Forum. <http://forum.newlc.com/>