Real-Time GSM Broadcast Receiver on a Cortex-M4 Microcontroller

Stefan Erhardt [#], Felix Pflaum [#], Robert Weigel [#], and Alexander Koelpin [*]

[#]Institute for Electronics Engineering, Friedrich-Alexander University of Erlangen-Nuremberg
Cauerstrasse 9, 91058 Erlangen, Germany, Email: stefan.erhardt@fau.de
[*]Chair for Electronics and Sensor Systems, Brandenburg University of Technology,
03046 Cottbus, Germany

***Abstract*— Recent microcontrollers provide enough computing power to perform signal processing while still being energy efficient and enable more flexibility than FPGAs. A fully functional GSM broadcast receiver was implemented on a Cortex-M4 microcontroller that is able to decode broadcast control channel (BCCH) messages. With several algorithmic optimizations it is able to operate in real-time in combination with a Sub-GHz transceiver as RF front-end. The timing was proven by measurements with an logic analyzer.**

I. Introduction

In an earlier publication a GSM broadcast channel receiver based on an ultra-low power Sub-GHz transceiver was introduced [1]. It was shown that the transceiver can be operated in a legacy mode for receiving and demodulating GSM signals, while the signal processing was conducted in MATLAB on a personal computer. For miniaturizing the receiver the complete processing chain should now be implemented on an ARM Cortex-M4 microcontroller.

Current state-of-the-art microcontrollers provide a 32-bit RISC core with floating point unit (FPU) and DSP instructions with a current consumption less than 100 µA/MHz while running at 80 MHz, thus enabling the opportunity of serious signal processing in mobile applications [2]. Still, hardware specifically optimized algorithms need to be found.

There are many possible applications for an ultra-low power GSM broadcast receiver: Within the Internet of Things it could be interesting to make use of the surrounding mobile network infrastructure, e.g. for a synchronization to a base station's clock that is typically derived from an expensive rubidium atomic clock. A clock-harvesting receiver was shown in [3]. For gaining true timestamps the broadcast channel of the base station must be fully decoded, therefore requiring extensive signal processing.

II. System Description

The test platform consists of an STM32L476 microcontroller with power supply, debugging interface and USB connection on one PCB and a Sub-GHz transceiver on an exchangeable PCB (fig. 2). The transceiver is configured to demodulate GSM downlink signals and recovers a clock

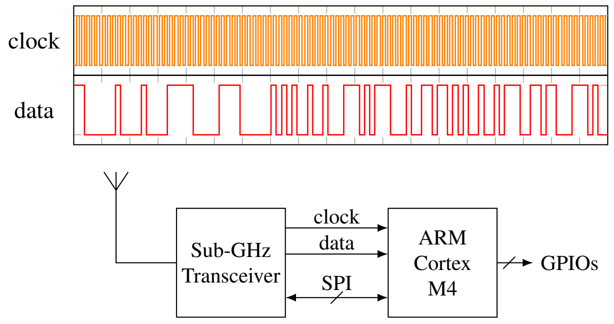

Fig. 1. Block diagram and input signals

and data signal, which is transfered to the microcontroller via GPIO pins (fig. 1). The signal processing for sampling, synchronization and decoding of the convolutional code is fully implemented on the microcontroller and runs stand-alone. For debugging and true time measurement purposes several GPIO pins are accessible.

First the synchronization channel (SCH) is searched by correlating with its training sequence and decoded by using the Viterbi algorithm. The gained data contains information about the timing within the multiframe structure. Finally, all four broadcast control channel (BCCH) bursts can be determined by correlating with their respective training sequences. After deinterleaving and decoding them, cell information data is available. A good reference for a deeper understanding of the GSM protocol depicts [4].

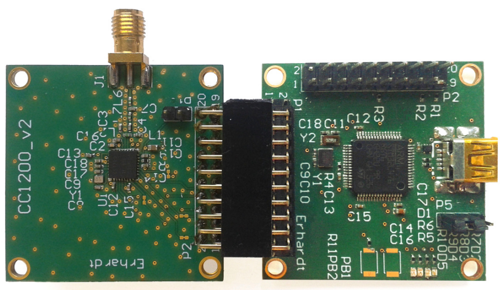

Fig. 2. Sub-GHz transceiver (left) and microcontroller (right)

III. Algorithms

A. Multiregister Shift Buffer

With every falling edge of the clock signal, an interrupt on the microcontroller is triggered and the respective IRQ handler is executed. Before reading in the new input bit, the buffer that contains the previous bits needs to be shifted by one, discarding the oldest bit. The shift buffer should have the size of one GSM burst, so the array must be of size $5 \cdot 32$ bit. The overflowing bits from each individual register bit shift must be preserved in the previous registers as can be seen in fig. 3.

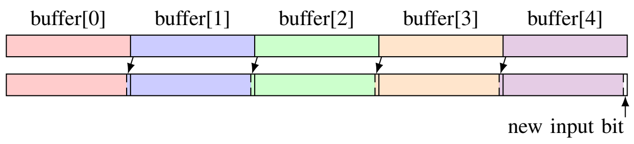

Fig. 3. Multiregister shift buffer

The ARM Cortex-M4 architecture only provides a right shift with carry instruction (RRX), whereas a left shift with carry is required. The solution is an addition of each value to itself, thus doing a multiplication by 2 or left shift by 1, and making use of the carry flag. In assembly language the full shifting process can be written as follows:

```
adds %r4, %r4
adcs %r3, %r3
adcs %r2, %r2
adcs %r1, %r1
adc  %r0, %r0
```

After the shifting process, the new input bit can be inserted into `%r4`. The registers can either be written back to the memory or directly used for the following correlation.

B. Simplified binary cross-correlation

Before being able to decode the desired downlink channel, a temporal synchronization has to be accomplished. Typically, a peak in the cross-correlation of the incoming bit stream with a known synchronization sequence is searched. For a discrete, complex input vector x and a correlation sequence s a cross-correlation y can be expressed by:

$$y[n] = \sum_{m=-\infty}^{\infty} x^*[m]\, s[m+n] \tag{1}$$

In order to calculate a positive and negative result, the binary inputs x, s need to be in the bipolar set $\{-1; 1\}$. With only real values and a finite s of length N, the correlation at the current position ($n = 0$) can be simplified as follows:

$$y = \sum_{m=0}^{N-1} x[m]\, s[m] \tag{2}$$

For easier implementation on the hardware the input vectors shall consist of elements in the unipolar set $\{0; 1\}$. The relation between bipolar and unipolar elements is considered with the substitution:

$$x_{\{-1;1\}} = 2 \cdot x_{\{0;1\}} - 1 \tag{3}$$

Furthermore, the multiplication can be substituted with an XNOR function, that only returns '1' if both inputs are identical:

$$y = \sum_{m=0}^{N-1} \left(2 \cdot \overline{(x[m] \oplus s[m])} - 1\right) = $$
$$= 2 \cdot \left(\sum_{m=0}^{N-1} \overline{(x[m] \oplus s[m])}\right) - N \tag{4}$$

Respectively, the cross-correlation result y can reach a value:

$$-N \leq y \leq N \tag{5}$$

After negating the XOR result for logical XNOR, the summation in binary arithmetics corresponds to counting the number of '1's or calculating the Hamming weight. In computer sciences this function is also called *population count* or *popcount*. The instruction set of Cortex-M4 does not provide a dedicated register popcount operation like x64 processors do. Fortunately, there is an optimized GCC implementation that takes twelve instructions on our architecture.

For peak detection an unipolar correlation result is sufficient. Thus the value range can be shrunk to unsigned integers by omitting the multiplication and subtraction:

$$y = \sum_{m=0}^{N-1} \overline{(x[m] \oplus s[m])}\,; \quad 0 \leq y \leq N \tag{6}$$

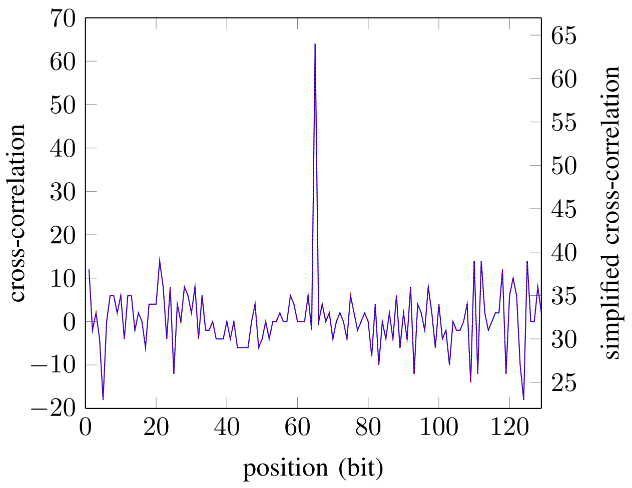

Fig. 4. Cross-correlation from Matlab (left axis) and simplified binary cross-correlation (right axis) of GSM input bit stream with SB training sequence.

The result of the simplified binary cross-correlation in comparison with the xcorr2 Matlab function is shown in fig. 4. Both curves are superimposable, only the axis scaling is different.

On a 32 bit architecture a bit-wise XOR operation requires only one clock cycle for the calculation of 32 bits. The training sequence of a SB has a length of 64 bit. According to (6) the summation can be split into $2 \cdot 32$ bit operations:

$$y = \sum_{m=0}^{31} \overline{(x[m] \oplus s[m])} + \sum_{m=32}^{63} \overline{(x[m] \oplus s[m])} \qquad (7)$$

Finally, a 64 bit simplified binary correlation can be written in C code with the buffer and the synchronization sequence being split up into arrays:

```
y  = popcount(~(input_buffer[0] ^ sync_seq[0]));
y += popcount(~(input_buffer[1] ^ sync_seq[1]));
```

C. Viterbi

The Viterbi algorithm is typically used for decoding convolutional codes by reconstructing the most likely encoded input. The algorithm consists of two steps: Calculate a forward state history matrix and then trace back the matrix, choosing the most likely, allowed path. Therefore, it utilizes the redundancy for forward error correction [5].

Initially, several look-up tables have to be set up: A $16{\times}2$ matrix that describes the state transitions of the encoder; a $16{\times}2$ matrix that contains the 2 bit outputs of the encoder; a $16{\times}16$ matrix that holds the originally encoded input bits. Additionally, the state history matrix has to be allocated in the memory. It has the size of the number of payload bits times number of states. For the SCH it results in $39{\times}16$, for the BCCH $228{\times}16$. Each matrix is a two dimensional array with data type uint8_t, e.g. the state history matrix consumes $228 \cdot 16 \cdot 8$ bit respectively 912 32-bit registers.

At the first part the state history matrix has to be filled with the surviving previous states. For that purpose the bit stream with a length of twice the payload bits is read in a loop in 2-tuples. In a second, inner loop every state is stepped through. For each state both possible inputs 0 or 1 are encoded and the Hamming distance aka popcount of this encoded output to the current received 2-tuple is calculated. Now the respective state transition is looked up in the state transition table and the previously calculated Hamming distance is saved as the so-called branch metric. Subsequently, the branch metric is accumulated in an array, since a certain state can have several possible predecessors. If the accumulated branch metric is cheaper than the value saved before, it is now saved into the branch metric instead and the state that lead to it is saved in the state history matrix. In the end of the first part, a fully filled state history matrix is available.

The second part is the trace back: Now the state history matrix that contains the surviving predecessor to each position is stepped through from its end to the beginning. The originally encoded message can be reconstructed by looking up the respective bit in the input matrix.

D. CRC and Deinterleaving

The decoded messages contain a CRC-encoded checksum to ensure data integrity. It is calculated by dividing the message with the respective CRC polynom in a loop, until the rest equals the polynom itself, if no bit error occurred.

Whereas the SCH can be decoded and CRC checked right after its appearance, the BCCH is both spread over four bursts and bit interleaved for higher robustness. Therefore, all bursts have to be received and deinterleaved before the Viterbi decoding starts. The deinterleaving can already be accomplished after the reception of each BCCH burst, filling up the data vector successively.

IV. Measurements

For all measurements the GCC compiler optimization was set to "optimize most" (-O3) and the instruction cache was enabled. The basic microcontroller initialization was conducted using the hardware abstraction layer (HAL) provided by ST.

A. Memory

The memory footprint can be evaluated with the tool arm-none-eabi-size that shows the code size in this format:

```
arm-none-eabi-size "project.elf"
text       data     bss     dec     hex  filename
11720       120    1952   13792    35e0  project.elf
```

'text' describes the code size in bytes. 'data' is used for initialized variables. 'bss' shows the size of global and static variables. 'dec' contains the sum of text, data and bss. 'hex' is the hexadecimal representation of dec.

By executing arm-none-eabi-size on *.o files, the footprint of single algorithms can be determined:

```
text    data      bss      dec      hex  filename
2368       0        2     2370      942  cc1200.o
 248       0        0      248       f8  shiftbuffer.o
1945       0       36     1981      7bd  gsm.o
1828       0        0     1828      724  viterbi.o
 280       0        0      280      118  crc.o
```

cc1200 contains all register settings and functions to simplify read and write to the CC1200 transceiver. shiftbuffer holds the GPIO interrupt callback routine and shift buffer handling. In gsm the simplified bit-wise correlation, the GSM bursts' bit mapping and the BCCH deinterleaving were implemented. viterbi contains the complete Viterbi decoder and crc two distinct functions for error checking the decoded SCH and BCCH data.

The basic GSM stack requires 6707 bytes. The remaining 7085 bytes are used by the microcontroller initialization and service routines.

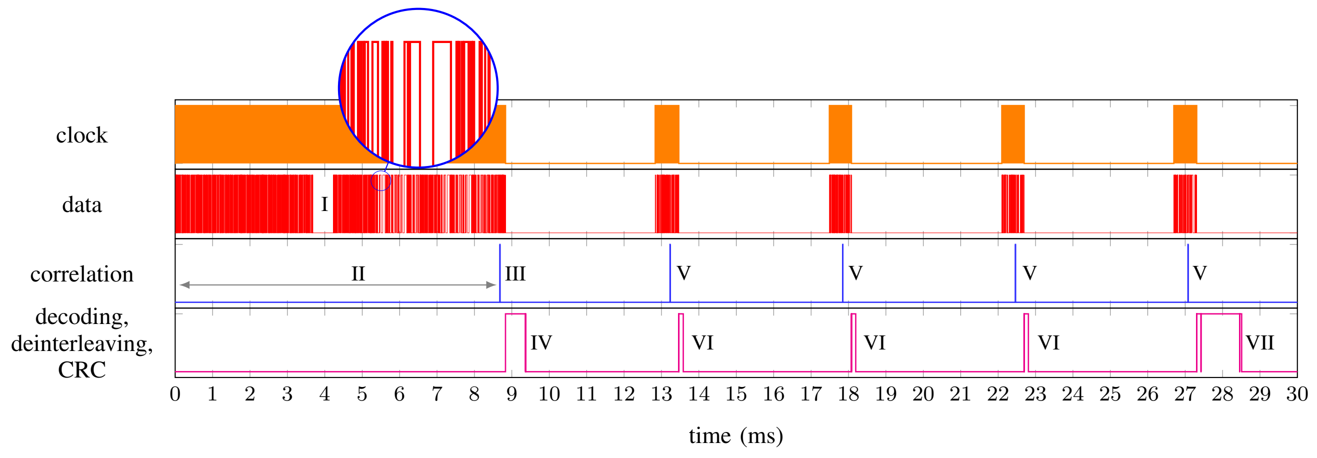

Fig. 5. Time signals tapped with logic analyzer

B. *Time measurements*

In order to determine the code efficiency, it is not sufficient to only count the number of machine code instructions, since reading and writing from or to the memory could require more time than one clock cycle, depending on the memory location. Therefore, true timing measurements were taken with a logic analyzer. A microcontroller pin was toggled at selected positions in the source code and sanity checked in the disassembly.

The time signals of clock, data and two GPIOs are depicted in fig. 5. The clock is a rectangular signal that displays as a solid box, since the time scale is too wide to resolve the signal. The receiver was switched off after synchronization and between the four BCCH bursts. Phases of interest are numbered and enlisted in the following table:

phase	description	duration
I	FCCH (one burst of zeros)	577 µs
II	Real-time correlation of SCH sync pattern	variable
III	SCH sync pattern found (correlation exceeds threshold)	-
IV	Decoding of SCH data and CRC checking	(534 + 4) µs
V	BCCH sync pattern found (correlation exceeds threshold)	-
VI	BCCH data deinterleaving	112 µs
VII	BCCH data deinterleaving, decoding and CRC checking	(112 + 1033 + 54) µs

In phase I the frequency correction channel (FCCH) contains a clearly visible set of 142 zeros.

In phase II and V a real-time correlation has to be performed in order to find the SCH/BCCH synchronization sequences. The clock signal fires interrupts with the GSM bit rate R, so the correlation needs to be executed at least within t_{max}:

$$t_{\text{max}} = 1/R = 1/277.833\,\text{kbit/s} = 3.60\,\mu\text{s} \tag{8}$$

Following times where measured:

- interrupt handler: 0.45 µs
- shiftbuffer: 0.40 µs
- correlation: 0.95 µs

Therefore, the total sync search time amounts to 1.80 µs and does not exceed the maximum time from (8).

V. Conclusion

It was shown that real-time digital signal processing for receiving GSM broadcast channels can be implemented on an ultra-low power Cortex-M4 microcontroller. A simplified binary correlation was mathematically derived from the definition of the cross-correlation and implemented on the target hardware. With optimizations in assembly code the calculation time for these operations could be undercut by half of the maximum available time. Furthermore, GSM broadcast messages can be decoded and checked by an viterbi decoder and CRC.

In future work the system could be miniaturized and optimized for energy consumption.

Acknowledgment

This work is funded by the German Research Foundation (DFG) under grant KO 4340/5-1.

References

[1] S. Erhardt, R. Weigel, and A. Kölpin, “Receiving GSM Broadcast Channels with an Ultra-Low Power Sub-GHz Transceiver,” in *2017 47th European Microwave Conference*, October 2017, pp. 380–383.

[2] STMicroelectronics, *Ultra-low-power ARM Cortex-M4 32-bit MCU+FPU (datasheet)*, 2017. [Online]. Available: http://www.st.com/resource/en/datasheet/stm32l476rg.pdf

[3] J. K. Brown and D. D. Wentzloff, “A GSM-Based Clock-Harvesting Receiver With -87 dBm Sensitivity for Sensor Network Wake-Up,” *IEEE Journal of Solid-State Circuits*, vol. 48, no. 3, pp. 661–669, March 2013.

[4] J. Eberspaecher, H. Voegel, C. Bettstetter, and C. Hartmann, *GSM – Architecture, Protocols and Services*, 3rd ed. Wiley, 2009.

[5] G. D. Forney, “The viterbi algorithm,” *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, March 1973.