Bachelor Thesis

A Secure Membership Service for Time-Triggered Protocols

Bernhard Urban lewurm@gmail.com

Sebastian Falbesoner sebastian.falbesoner@gmail.com

August 22, 2010

Supervisor: Univ.Ass. Dipl.-Ing. Dr.techn. Christian **El-Salloum**

Abstract

A membership service provides information about faulty nodes (e.g. nodes not synchronized with the global time) which is vital for safety-critical applications. FlexRay does not provide a membership service by design, hence it must be implemented on the application layer. Openness and connectivity of real-time systems have increased over the years, thus a logical step forward is to *secure* this service. Clock synchronization is the most elementary and therefore most vulnerable operation in a time-triggered protocol. By using a secure membership service, we also secure the concerning clocks. On top of that, other services can be realized on a higher layer.

We have implemented the secure membership service on a FlexRay evaluation board ("bitspot blue" by Fujitsu) by using two different approaches:

- symmetric encryption only (AES)
- symmetric encryption in combination with public key cryptography (RSA-AES)

We have analyzed both implementation variants with several parameters (key size and frequency) with respect to computation time and memory usage.

Contents

Preface

The thesis describes the development of a secure clock synchronization mechanism which was inspired by the fact that the FlexRay communication protocol lacks a *membership service*.

In Section 1 we impart the basic knowledge needed to understand the core of the thesis by introducing the important term "real-time". We will see that there are, according to [Kop97] three classifications of real-time systems and that in the safety-critical one, only systems with a hard deadline are relevant. The two basic paradigms for real-time communication protocols, namely the *event-triggered* and *time-triggered* approaches are discussed and compared. The result is that the TTA (time-triggered architecture) is in many ways more reasonable with regard to determinism and timeliness. Its counterpart, the event-triggered architecture, potentially leads to many more problems, one of them being the absence of *idempotence*. The meaning of time and common clock synchronization methods are discussed to get an idea why clock synchronization is important. We will see that a combination of *rate correction* and *offset correction* leads to the best results. The section is concluded with a few examples showing why security can become safety relevant and it can be inferred from the discussion that global time has to be secured.

Section 2 focuses on FlexRay, one of the most used real-time communication protocols in the automotive industry nowadays. After the introduction of the history of the protocol and the listing of some examples of cars that use FlexRay, the capabilities and technical details are mentioned. We will see that it does not only support high-speed communication and basically follows the time-triggered architecture, but also allows event-triggered communication. The big drawback of FlexRay however – that has led to the writing of this thesis – is the fact that it does not support a membership service.

The section is resumed with a listing of the most important configuration parameters, with example values shown on the BMW 7. In a final step, the details of the clock synchronization algorithm that is used within FlexRay is examined more closely. It is about a *Fault Tolerant Midpoint Algorithm* which is very easy to implement in hardware and always converges.

In Section 3, which is the core of the thesis in connection with Section 4, the development of such a membership service is described in an abstract way, that is to say without depending on any specific computer architecture. For simplifying the process we have used a *master-slave* model for our secure membership service. As this leads to a *single point of failure*, an important possibility to improve our implementation would be the development of a purely distributed system which does not depend on one single master node any more. This would however have gone far beyond the scope of this thesis and project practical. The cryptographic algorithms that are used in our work, namely AES for symmetric encryption and RSA for public-key encryption, are also briefly explained by describing which role they play in our model. This section especially emphasizes the fact that the secure clock synchronization is achieved through the secure membership service.

Section 4 discusses the details of our implementation and can be partly regarded as the documentation for it. It includes concrete C source code for our target architecture, the *FR60 MB91F465X* microcontroller by Fujitsu. The used hardware, tools and libraries are explained and the problems we encountered while trying to embed them in our project. In the end, a performance analysis is figured, containing values with varying key sizes and execution frequencies.

Acknowledgments

Christian El-Salloum for supervising our bachelor thesis and two practical training projects.

Iris and Franz Winkelbauer, Josef Falbesoner, Bernhard Bliem, Marie Utkina for proofreading.

Developers of VImproved, `git`, GNU Make, TEX Live, LATEX, KOMA-Script, PSTricks and `highlight`, although they will probably never read this.

1. Introduction

1.1. Real-Time Systems

The term "real-time" is used in computer systems in which the temporal component plays a major role; that means that the correctness of the system behaviour depends not only on the logical results of the computation, but also on the physical time, when these results are produced.

In many areas of computer science research, however, the temporal component is not of any importance at all, e.g. in compiler construction. Of course, it is always desirable to achieve high performance and to minimize latency for improving user-friendliness, but no matter how long it takes a compiler to translate the source code into the binary, the result will still be correct. In real-time systems such as automotive systems, after a certain point in time – the so-called *deadline* – the result of the computation process has no utility or, even worse, a catastrophe could be the result. A computer system in an automobile for instance has to ensure to react in a given time on the input by the driver (e.g. the braking-pedal) regardless of whether there is currently a high load on the system or not. Typical deadlines in real-time computing are in the range of milliseconds (ms) and microseconds (μs). The deadlines of real-time systems can be classified into the following categories [Kop97]:

- **soft** deadline: The result has utility even after the deadline is missed, but does not cause serious damage to its environment. (not the focus of this thesis)

  Example: Consider a simple task of a computer system in an automobile that checks the oil level of the engine every ten seconds. If, for any reason, the result can not be delivered to the dashboard in time (say 100 milliseconds), this missing of time has no harmful consequences.

- **firm** deadline: The result has no utility after the deadline has passed, but does not cause serious damage either.

  Example: A packet of a video stream is too late due to a different route through the IP-network. Thus, the screen flickers for a short moment, yet this is nothing but an annoying consequence.

- **hard** deadline: Missing a strict deadline can cause a catastrophe.

  Example: If the computer system of the automobile in the example above can not fulfil the timing requirements for the braking subsystem (e.g. anti-lock braking system or electronic stability control) this could result in an accident and injure or even kill people who are involved.

  Other examples are aeroplane control systems, car engine control systems and heart pacemakers.

  Such systems, that include at least one hard deadline, are called *hard real-time systems*, which are discussed in this work.

1.2. The Time-Triggered Architecture

Designing a communication protocol for hard real-time systems is by no means trivial and requires some special consideration of how to guarantee the reception of messages before the deadline.

Generally it is important to ask the question: at *which point in time* it is allowed to send messages? Is it wise to send them as soon as possible when requested? This seems quite logical and, in Ethernet for example, this is the case – whenever a host wants to send a packet, this is tried immediately. However, this freedom of being allowed to send whenever you want of course implicates the possibility of collisions on the bus. In the OSI Reference Model there is a special layer (2a) for avoiding these collisions, called *Media Access Control* (MAC). In case of Ethernet this is handled via the *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD) mechanism: whenever a host detects that there is already another host sending on the bus, it waits for a random delay and tries again. It is quite clear that a system constructed like this is not suitable for real-time communication, because it is not predictable when a message arrives. With the help of statistics of course, you can determine the probability of collisions and thus the probability that a message arrives within the deadline. But even this method largely depends on the other nodes: if due to a failure a host tries to send packets in an endless loop, the bus is disturbed and all messages of other nodes will probably never arrive.

The mechanism described above falls into the category of *event-triggered* architecture. A widespread event-triggered protocol that has been de-facto standard for years in the automotive industry is the *Controller Area Network* (CAN) bus, which uses a MAC mechanism similar to Ethernet.

A different approach, originally designed at the Technical University Vienna in the early 80s is the *time-triggered* architecture: as the name indicates, the only factor responsible for sending/receiving messages is the progression of time. There exists a specific "roadmap" that determines when exactly each node can send and receive to avoid collisions. For this purpose every communication round is divided into slots; the mechanism is called *time division multiple access* (TDMA). Thus, every message of each node is *a-priori* known when it is sent in a given communication cycle. In Figure 1, a TDMA scheme is diagrammed with the setup of a communication cycle that consists of four slots.

In order to be sure that a node does not send on the bus when it is not allowed to do so (this could happen to faulty nodes like the typical "babbling idiot"), in some implementations of the time-triggered architecture there are even *bus guardians* that avoid sending on certain slots physically.

The advantages of the time-triggered approach for hard real-time systems are obvious:

- Because of the static round scheme the problem of media access control is implicitly solved and it is guaranteed that messages arrive within the deadline. As a bonus, the sender and receiver addresses need not be included in a message frame since this is implicitly determined by the "roadmap"

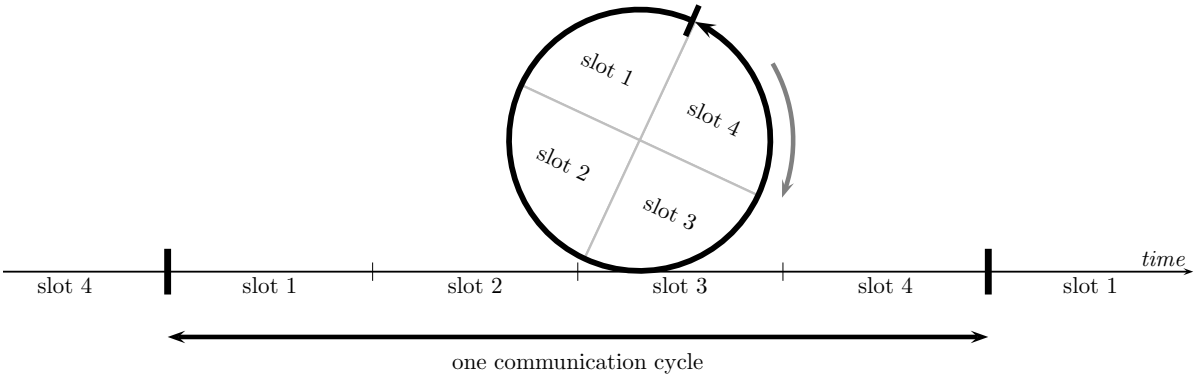


Figure 1: Illustration of a TDMA Communication Cycle

- It is not possible that the bus is under (uncontrollable) "high-load" any longer, because the performance on the bus is determined before the system is even executing and it never changes
- In contrast to event-triggered messages, which follow an *exactly-once semantic*, it is not a problem if a message gets lost within a round because it will be sent in the next round anyway. Furthermore a receiver is not forced to process a receiving message immediately. This property is called *idempotence* [Kop97]: State-messages are idempotent, that means, receiving more than one message of the same kind has the same effect as receiving a single one.

One problem in the time-triggered architecture however is the fact that a global view of the term *time* has to be ensured in a distributed system. This is described in detail in Section 1.3.

The most commonly used TTA implementations nowadays are the *Time Triggered Protocol* (TTP/C), invented at the Technical University Vienna[1] and *FlexRay*, a protocol that was designed with the main focus on the automotive industry, which will be described in detail in Section 2.

Example: Temperature Sensors in a Factory
In order to expose the benefits of a time-triggered architecture over event-triggered architectures regarding load behaviour, we provide a simple example: Assume you have four nodes in a factory connected to a communication bus (bandwidth of 100 kbit/s), in which each node is connected to 16 temperature sensors. Additionally a node to supervise the sensors is connected. Once a configured threshold of a sensor is exceeded, its associated node sends a message over the bus to the supervisor. This has to happen within ten milliseconds.

Note: Both solutions use the same protocol, e.g. CAN. To simplify calculations suppose the message-header has a fixed length of 40 bits.

[1]since 1998 TTech Computertechnik AG has taken over the further development of TTP

Event-triggered solution A node sends a message as soon as one of its sensors reaches the given threshold. To identify a specific sensor, the sensor-number has to be encoded into the message:

$$log_2(4 \text{ nodes} \cdot 16 \text{ sensors}) = log_2(64) = \underline{6}$$

bits are needed. This results in a message-size of $40 + 6 = 46$ bits. Therefore, in ten milliseconds at a bandwidth of 100 kbit/s about

$$\lfloor \frac{100 \cdot 10^3 \cdot 10 \cdot 10^{-3}}{46} \rfloor = \lfloor \frac{1000}{46} \rfloor = \underline{21}$$

messages can be transmitted punctually. But in a worst-case scenario, up to 64 messages have to be sent simultaneously, which can not be carried by the event-triggered solution.

Time-triggered solution Consider a TDMA-like manner, in which each node sends a message with *every* communication cycle in its slot. A message contains the header (40 bits) and a 16 bits boolean array, so there is one bit for each sensor connected to the node. Since a node sends its message on an a-priori point in the communication cycle, no data for identification has to be transmitted. Thereby, in ten milliseconds at a bandwidth of 100 kbit/s about

$$\lfloor \frac{100 \cdot 10^3 \cdot 10 \cdot 10^{-3}}{40 + 16} \rfloor = \lfloor \frac{1000}{56} \rfloor = \underline{17}$$

messages can be transmitted in time. Compared to the event-triggered solution these are slightly fewer messages, however, you have to consider that in one message 16 sensors are handled at once. Thus, in a worst-case situation only four messages have to be transmitted, which can be easily achieved.

The load on the bus by both solutions is illustrated in Figure 2.

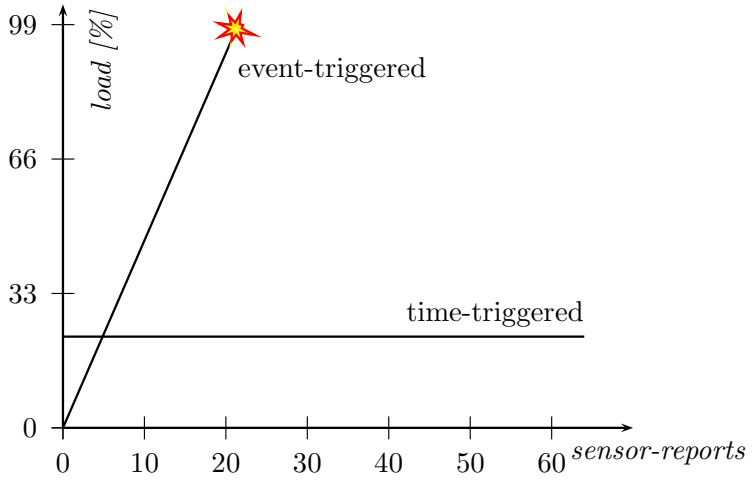


Figure 2: Event- vs. Time-triggered Solution

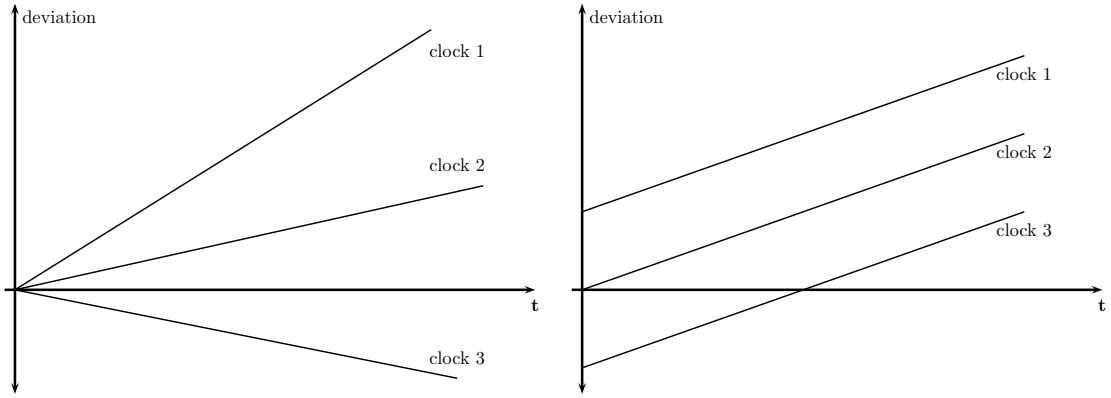
1.3. Why is Clock Synchronization important?

Due to the fact that the time-triggered architecture is based merely on the progression of time, it is crucial that all nodes have a consistent view of the current time. For that purpose the term *global time* is introduced, which is achieved via a *clock synchronization* mechanism.

1.3.1. Why do Clocks deviate?

Nowadays clocks are characterized by an astonishing precision, however even atomic clocks exhibit deviations of the "perfect time" after some time. Taking this into consideration, the clocks used in embedded systems[2] have – when compared to watches, wall-clocks or even atomic clocks – quite large inaccuracies, because the devices are quite simple and are occasionally exposed to extreme environmental conditions.

A clock is generally characterized by two properties, namely frequency and phase. If two clocks have exactly the same value in both of these parameters, they are said to be synchronized. There can be two reasons why clocks are not synchronous to each other. If the *frequency* of clocks differs, the deviation of the clocks to each other will increase by time. Figure 3(a) shows an example of three clocks that are started simultaneously. If – in another extreme example – the clocks start with a different *phase*, but with exactly the same frequency, the deviation between the clocks is constant (see Figure 3(b)).



(a) different frequency, but same phase (b) different phase, but same frequency

Figure 3: Several Clocks with different Settings

In embedded systems, oscillators are used as frequency source for a clock. Even in high-quality silicon oscillators there are deviations to their actual normal frequency. Reasons for these frequency deviations include manufacturing tolerances, variations in temperature, aging and vibrations.

[2]nowadays mostly realized by timers directly integrated in the microcontroller

Mathematically spoken, the drift rate of a clock k is defined as follows [Kop97]:

$$\rho_i^k = |\frac{z(\text{microticks}_{i+1}^k) - z(\text{microticks}_i^k)}{n^k} - 1|$$

where

- $z(...)$ denotes the perfect reference clock which returns a timestamp of the given event.
- microticks_i^k is the i-th microtick by clock k. A microtick is the smallest step a clock k can do (which is in turn clock specific).
- n^k is a nominal number of ticks of the reference clock z within a granule of clock k.

The perfect reference clock has a drift rate of $\rho^z = 0$ [s/s], whereby real clocks have a drift rate between 10^{-2} and 10^{-8} [s/s]. In practice a drift rate results from two properties of real clocks, namely frequency and phase as has already been discussed above.

1.3.2. Clock Correction

There are two fundamental methods to correct the deviation of time of a clock which form the base of a stable clock-synchronisation algorithm [Rau07]:

offset correction The counter value of the clock is modified directly. This method needs an external clock, whose time value is used for synchronisation. Offset correction solely works upon the symptom of the clock deviation; the real cause of the frequency deviation of the oscillators is not corrected, so after some time the corrected clock will have a deviation to another clock again.

rate correction This method tries to solve those problems. The oscillator frequency is usually divided into a lower frequency by using prescalers, before the "time is counted". Of course the frequency of an oscillator can not be changed directly, but the divider ratio can be adapted according to the frequency deviation. In this way, the resulting frequency can be accelerated or slowed down. Therefore, the relation between input and output frequency is influenced by the rate correction. The advantage of this method is the fact that once clocks are synchronized they do not deviate that much over a long period of time.

Of course, it is not surprising that a combination of both methods, offset and rate correction, leads to the best results.

1.3.3. Clock Synchronization

The goal of a clock synchronisation algorithm in a real-time system is to ensure that the *local* clock of each correct node in an ensemble is within a given precision Π [Kop97]. The precision at an instant i of the omniscient clock is defined as following:

$$\Pi_i = \max_{\forall\, 1 \leq j,k \leq n} \{\text{offset}_i^{jk}\}$$

whereas the offset between a clock j and a clock k at a point in time i is defined as:

$$\text{offset}_i^{jk} = |z(\text{microtick}_i^k) - z(\text{microtick}_i^k)|$$

where $z(...)$ and microtick_i^k are defined as above.

One example of such an algorithm which fulfils those conditions is the *Fault Tolerant Midpoint Algorithm* (see Section 2.3 for more details on it).

1.4. Security can become Safety relevant

A common problem of popular protocols is that they were not intended for security applications *by design*. Take the *Domain Name System* for example: It has several security issues, like DNS cache poisoning to name one, because its designers had no or little security in mind.

So have time-triggered protocols, such as TTP and FlexRay. Although they have been designed for applications which are safety relevant, one has to consider that safety is not equal to security. The reverse conclusion however is possible: As intruders might try to manipulate global time, security can become safety relevant. In a common definition in computer science [RG91] the term "security" describes three distinct aspects:

secrecy / confidentiality information has to be disclosed to anyone who is not authorized to *access* it

integrity any unauthorized malicious or accidental *changes* of information must be avoided

availability the computer system's hardware and software must keep working efficiently (must be *available*) and should be able to recover quickly and completely if a disaster occurs

In usual (non real-time) computer systems secrecy seems to be the most important aspect – some people do not even think of the other two aspects and put security on a level with secrecy, e.g. by keeping their critical e-mails secret with some kind of encryption. With regard to hard real-time communication systems themselves, secrecy however is the least important of the three aspects. If you take a car for example, most of the information that is sent through the protocol is control information anyway, and a "bad guy" who wants to exploit the system will not have much benefit by intercepting the bus and thus being able to watch data frames containing actual sensor values or

motor and climate control data. In this case the guarantee of integrity and availability are far more important – if the intruder succeeds in manipulating the control data, e.g. by constantly overwriting brakes control data, and injecting motor control data, which leads to acceleration, any attempt to slow down the car will fail ultimately and the drive will end deadly. A crafty terrorist could manipulate flight control data in an aeroplane to provoke an accident that claims hundreds of victims – to mention an even more drastic example. Compromised real-time systems in nuclear power plants could of course also lead to catastrophic accidents. All those examples illustrate how the *integrity* aspect of security could lead to safety problems: It is obvious however that all this is also closely connected to *availability* – if an important node in a real-time communication network is just down, the whole system is compromised. Availability can be improved via redundancy, but this is not the subject of this thesis; this work concentrates on developing a membership mechanism[3] that supports the avoidance of *global time* manipulation.

1.5. Global Time has to be secured

If the intruder manages to manipulate global time by drifting away the clock count of a certain node, the results could be catastrophic as well because the data frames (which themselves remain unmodified) arrive at a wrong instant and could do something that is not intended. For this reason it is important to have a mechanism that detects those manipulated nodes and puts them into a safe state. Of course, the drawback is that some of the bandwidth of the communication cycle needs to be statically reserved for a secure membership implementation. As it uses cryptography algorithms (e.g. AES or RSA) for increasing security, this also increases the CPU load because of the permanent encryption/decryption procedures, but in safety-critical systems a small loss of bandwidth and performance is usually accepted as the avoidance of accidents has top priority.

[3] see next section for a description of what membership is in terms of real-time communication systems

2. The FlexRay Protocol

2.1. History

Before the appearance and development of FlexRay in 2000, the standard communication system in the automotive industry was the CAN-bus, developed by Bosch in 1983. It followed a purely *event-triggered* approach and is thus not suitable for hard real-time systems. Potential bus collisions are resolved with *Carrier Sense Multiple Access with Collision Avoidance* (CSMA/CA): whenever two hosts try to send simultaneously, the one with higher priority is allowed to send, and the other has to wait. This approach is better than the MAC mechanism used in Ethernet (as described in Section 1.2), but it still leads to problems when a faulty high-priority node becomes a "babbling idiot" and the other lower-priority nodes come down with starvation and the bus is blocked.

Another bus system widely in use is *Local Interconnect Network* (LIN), which is used for a cost-effective communication between sensors and actuators and other systems that do not need to operate in real-time and do not need much bandwidth. As it is a master-slave system, there is no collision detection; all messages are initiated by the master. Typical applications are the networking in a door or seat, light control, climate control etc.

Other protocols that were used most notably in the BMW series were the byteflight protocol (BMW 5, 6 and 7 series) and later Time-Triggered CAN (since 2007 in the newer models of BMW 5 and 6 series). The former was developed especially for use in passive safety systems like airbags and can be seen as the predecessor of FlexRay; it already took advantage of TDMA for realizing a deterministic real-time behaviour.

The FlexRay Consortium

In the year 2000, the concerns BMW, DaimlerChrysler, Motorola and Philips founded an industry consortium with the primary goal of developing a fast communication system for the special requirements in the autocar. By now, more concerns have joined the FlexRay consortium, so since the year 2003 it is made up of the following well-known core members:

BMW

Daimler AG formerly (until 2007) DaimlerChrysler AG

Freescale Semiconductor spin-off from Motorola in 2004

NXP Semiconductors founded by Philips in 2006

Robert Bosch GmbH joined 2001

General Motors joined 2002

Volkswagen joined 2003

The first series production vehicle was the *BMW X5* at the end of 2006. The first autocar that would *fully utilize* the FlexRay system was introduced in 2008 in the new *BMW 7 Series (F01)*. The new Audi A8 (*2011 Audi A8 4.2 FSI*) that will be on sale in November 2010 will also use FlexRay, connecting approximately 30 ECUs [Dav10].

2.2. Goals and technical Details

Besides economical goals concerning cost advantage and the avoidance of compulsory licence fees, the most important technical goals that were realized in *FlexRay* are the following [Rau07]:

- **bus speed of 10 Mbit/s**
  The usual CAN data rate of 500 kbit/s is exceeded by a factor of 20, so that you have a reserve for the next few years.

- **redundant communication channels**
  With the use of two physical communication channels it is possible to either have redundant communication to reduce failure probability or to double the bandwidth (to 20 Mbit/s gross).

- **supply of a global synchronized time base**
  Having a global time base is an absolute requisite for an implementation of the time-triggered architecture.

- **guaranteed message transmission**
  Experiences with CAN have showed that messages get lost sometimes and that there are large variations concerning the arrival of messages, due to the event-triggered approach. By using the time-triggered architecture, those problems can be eliminated.

- **support for both time- and event-triggered communication**
  By splitting up the communication cycle in a static and a dynamic segment, event-triggered communication for applications that are not real-time relevant can be established.

- **compatibility to the byteflight protocol**
  Many existing components like bus drivers, star couplers, development boards and tools for the byteflight protocol as well as experiences with handling it have caused that decision. By now, the compatibility was dropped because this request would have limited the protocol way too much and other development goals would not have been achievable. The minislotting-technique used in byteflight however has been adapted for the dynamic event-triggered part of FlexRay.

As shown in Figure 4, a FlexRay node consists of a microcontroller (the *host*), the FlexRay-controller itself – called *communication controller* (CC) – and one or two bus

drivers[4]. The bus drivers realize the physical connection with the communication channel; FlexRay supports electrical as well as optical media. The FlexRay-controller implements the (logical) FlexRay-protocol. On the host, the application that sends and receives messages from or to other nodes is running. Before communication can be established, the host has to configure the CC.

With regard to media access control, FlexRay is basically a combination of TTP/C and byteflight – the former uses the TDMA mechanism and is thus used in the static segment, the latter uses mini-slotting for the dynamic segment, supporting event-triggered communication. A typical communication cycle is illustrated in Figure 5. Besides of the static and dynamic segment, there are also the *Symbol Window* and the *Network Idle Time* (NIT) in a communication cycle, whereby only the static segment and the NIT are obligatory. The Symbol Window is a special time slot that is reserved for symbols, e.g. for waking up all nodes in the network. During the last part of the cycle, the NIT, there is no communication on the bus - it is used for controller-internal processes and for clock synchronization.

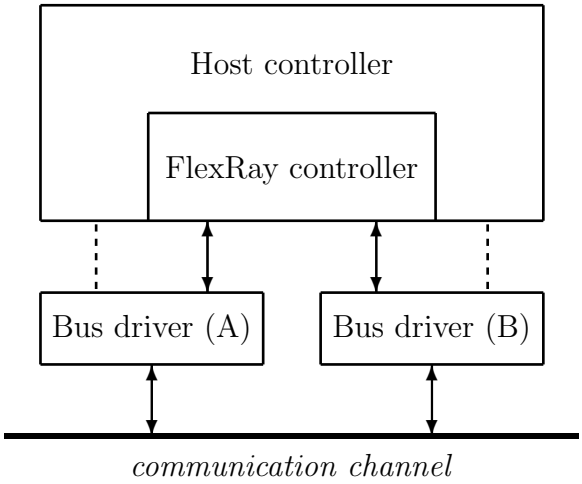


Figure 4: Schematic of an integrated FlexRay Node

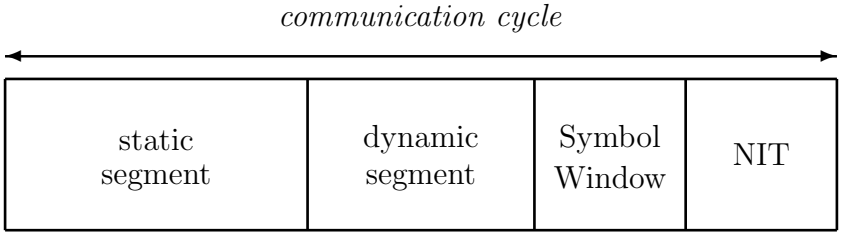


Figure 5: The FlexRay Communication Cycle

In the static segment there are slots with a fixed length and every node can take use of one or more of those slots to send or receive messages. Note that this allocation must be set in the configure state of the FlexRay controller and can not be changed during the running phase. Messages are called *frames* in the FlexRay terminology. Every one has to be identified with a certain *Frame-ID*. In slot 1, only frames with Frame-ID 1 are sent, in slot 2 only frames with Frame-ID 2 and so forth. Slots can also be empty, which can be useful for future extension or for the application to have some time to perform necessary calculations. The layout of a frame is illustrated in Figure 6.

[4]note that there are also stand-alone FlexRay-controllers which are connected to the host via address and data bus, the integrated approach however is more common because of cost advantages and faster exchange of data

The header section of the frame contains certain control bits, the Frame-ID, the payload length, a header CRC and the current cycle count. Note that the number of bytes in the payload section can only be even (the payload length in the header is doubled to determine the actual size). Thus the set of all possible total frame sizes can be described as: $s = \{x \in N \mid x \geq 8 \wedge x \leq 262 \wedge x \ mod \ 2 \equiv 0\}$.

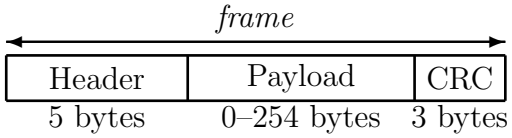


Figure 6: Layout of a FlexRay Frame

The mini-slotting technique used in the dynamic segment is not that relevant for hard real-time systems and is thus not covered in this short FlexRay introduction as it goes far beyond the scope of this thesis. See [BMR00] for a detailed description on byteflight and its MAC mechanism.

FlexRay uses a *Non Return to Zero* coding to shape the bitstream into physical form. Additionally, before every data byte the two bits **10**, called *Byte Start Sequence* (BSS), are inserted. Thus, for the transfer of one single byte, 10 bits are used physically. The falling edge between high- and low-bit is used for bit-resynchronization on the receiver.

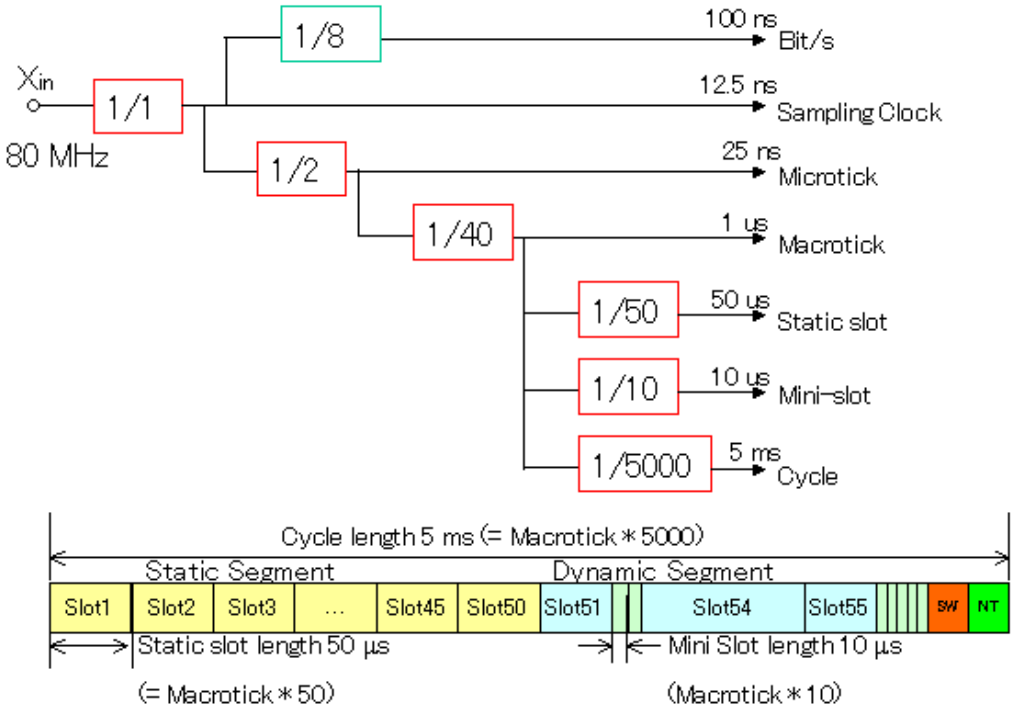


Figure 7: Temporal Configuration Values and their Dependencies [Mic10a]

There are lots of parameters (see [Con05]) that need to be configured in FlexRay, the most important being:

gdMinislot Duration of a minislot.

gdStaticSlot Duration of a static slot.

gMacroPerCycle Numbers of macroticks in a communication cycle.

gNumberOfMinislots Number of minislots in the dynamic segment.

gNumberOfStaticSlots Number of static slots in the static segment.

gPayloadLengthStatic Payload length of a static frame.

gdNIT Duration of the Network Idle Time.

pMicroPerCycle Number of microticks in the communication cycle of the local node. This value can differ from node to node since a microtick is clock specific.

Figure 7 shows the dependencies of some temporal configuration values in a sample FlexRay setup. Figure 8 lists setup values of the BMW 7 FlexRay communication network.

Parameter/Merkmal	Wert
Datenrate	10 Mbit/s
Zykluszeit	5 ms
Statisches Segment	3 ms
Dynamisches Segment	2 ms
Nutzdaten statisches Segment	16 byte
Nutzdaten dynamisches Segment	2 bis 254 byte
Statische Slots	91
Minislots im dynamischen Segment	289
Dauer statische Slots	33 µs
Dauer Minislots	6,875 µs
Dauer Makrotick	1,375 µs
Aktive Sterne	1
Maximale Kabellänge	24 m
TSS-Verkürzung	7,5 bit
Quarztoleranz	±200 ppm

Figure 8: FlexRay Configuration Values of the BMW 7 [BPS08]

2.3. Clock Synchronization in FlexRay

In FlexRay no reference clock for global time exists. Thus a clock synchronization algorithm is required to establish its own local view of the global time for each node. Since the configuration is done off-line, all time relevant settings are known a-priori. As

a consequence, a node can determine a specific deviation to another node by comparing the timestamp of a received frame with the expected timestamp and calculate the clock correction data out of them, which is done by a clock synchronization algorithm. For that purpose FlexRay uses a *Fault Tolerant Midpoint Algorithm* (FTMA). The advantage of that algorithm is that it is easy to implement in hardware, because besides of sorting and selecting the values the only CPU operations needed are *"addition"* and *"division by 2"*. It is proven that this simple algorithm always converges and one or two defective measurement values (depending on the number of available measurement values) do not influence the result.

The calculation of the "midpoint" of a measurement series is done by performing the following steps [Rau07]:

1. The measurement values are sorted
2. The k largest and k smallest values from the list are discarded
3. The largest and the smallest value from the remaining list are selected
4. Those two values are added and divided by two

The value k, which determines how many values are withdrawn from the list, depends on the number of available measurement values:

number of values	k
1-2	0
3-7	1
>7	2

Because of a much simpler implementation, the algorithm only uses integers. The critical part considering the need of floating-point arithmetic in this algorithm is the division by two; whenever an odd number is divided there is a remainder and the exact result would contain a decimal place ($x.5$). In this case it will always be rounded towards zero, i.e. a positive number will be rounded down and a negative number rounded up.
For example, $\frac{23}{2} = 11$ and $\frac{-23}{2} = -11$.

3. A Secure Clock Synchronization Algorithm

By using a secure membership service, the local view of global time is encrypted with a cryptography algorithm and sent to a trusted master node. The master node compares the content with its *own* view of global time and generates a membership vector with this information. Therefore we gain an advantage implicitly, namely a *secure clock synchronization*.

3.1. Attacker Model

Before we discuss the algorithm in detail, we want to propose an *attacker model* as motivation for a secure membership service. The model describes the different approaches how a potential attacker could manipulate the behaviour of the system to his favour. There are generally two types of attacks that can be thought of:

- **Delay / acceleration attacks**
  In real-time systems, the semantic of a message depends not only on its content, but also on the point in time, when it arrives at the receiving node. By physical modifications on the bus, the attacker can influence the duration for a message transmission. One could cut the bus line to a node and place a electrical device in-between which does nothing other than delaying the incoming signal. This is called a *delay attack*. The opposite of that, which is surely harder to achieve because the attacker needs to shorten the bus (which is often microscopical small nowadays) somehow, is named *acceleration attack*. Delay and acceleration attacks can not be avoided themselves, but in most cases they are useless anyway, because the clock synchronization mechanism will notice that the corresponding node is out of sync.

- **Time drifting attacks**
  A more threatening approach, which is based on the delay and acceleration attacks, is drifting away the global time. The idea of *time drifting attacks* is to trick out the synchronization algorithm which will not notice that something is wrong as long as the global time view changes only in very small steps. Time drifting attacks also can not be avoided themselves, but through our membership service all slaves receive a vector that shows which of the other nodes tried to be evil by trying to drift the global time away.

For understanding the thesis and the meaning of the secure membership service it is vital to consider the fact that both of those attacks can not be avoided. They are performed on a very low level and can not be fighted with common hardware or software security measurements, but instead the whole system – all nodes and the connecting bus lines – needed to be guarded in a physical way, which would increase costs and effort and is often not even possible. The purpose of our secure membership service is only to *detect* possible attacks by supplying every (slave) node with a vector that shows which of the other nodes are still trustworthy.

3.2. A Secure Membership Service

The secure membership service implementation is basically a master-slave system: Only one node in the system (the *master*) receives the time information of the slave nodes regularly and constructs the *membership vector* out of it. This vector is just an array of elements, for each node containing either 1 ($\rightarrow$ slave is alive and its global time is synchronized) or 0 ($\rightarrow$ slave is not alive or its global time is compromised). For N slave nodes, the membership vector $\overrightarrow{m}$ can be described mathematically as:

$$\overrightarrow{m} = \begin{pmatrix} m_0 \\ m_1 \\ m_2 \\ \vdots \\ m_{N-1} \end{pmatrix}, \; m_i \in \{0, 1\}, \quad |\overrightarrow{m}| = N \tag{1}$$

The number of "healthy" nodes in a network n_{healthy} can be calculated simply by adding all the elements of the vector:

$$n_{\text{healthy}} = \sum_{i=0}^{N-1} m_i \tag{2}$$

Logically, the difference to N results in the number of faulty nodes, n_{faulty}:

$$n_{\text{faulty}} = N - n_{\text{healthy}} = N - \sum_{i=0}^{N-1} m_i \tag{3}$$

Every node is able to get the current global time – or, to be more specific, its *own view* of the global time – by calling a certain function from the FlexRay driver, which we will call `getGlobalTime()` from now on. The service works the following way (compare with Figure 9):

1. **Slaves** - at the start of each cycle: call `getGlobalTime()`, encrypt the result and send it to the master
2. **Master** - receive the encrypted time information and decrypting it
3. **Master** - construct the membership vector out of all received slave timing information, sign it and send it to the slaves
4. **Slaves** - receive the membership vector; if the slave detects that it is out of membership, it turns into a safe state

Furthermore the secure membership has several properties:

Global Time is sent from each slave to the master. This was primarily introduced in order to prevent replay attacks.

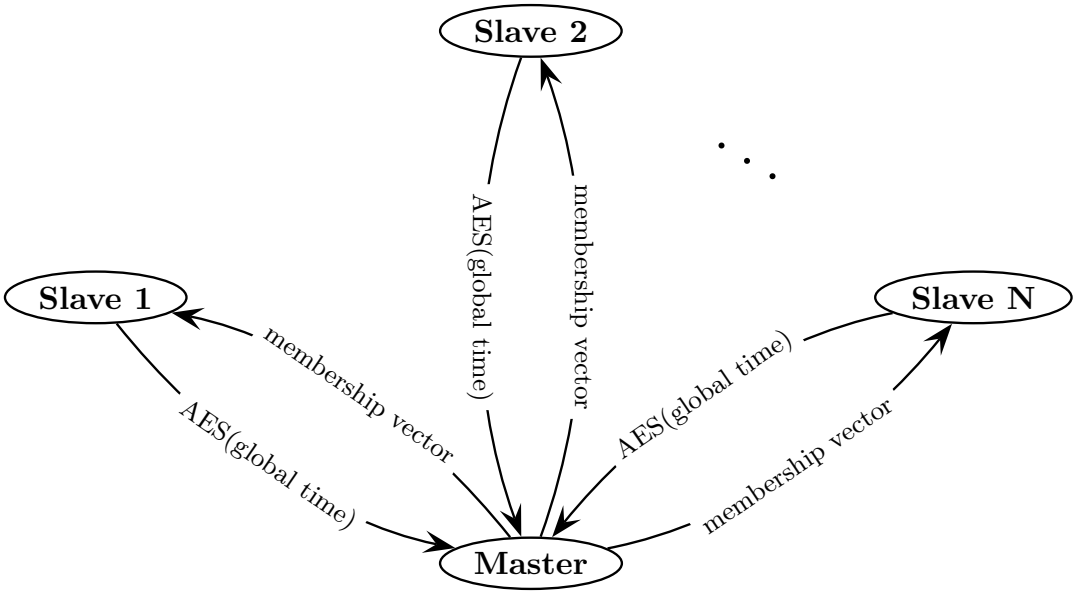


Figure 9: Visualization of the Membership Process

Reducing Overhead by discarding unneeded information of the global time data structure. In FlexRay for example the *Macrotick Value*, which is the current offset in a communication cycle, can be discarded because the instant of the receiving is already known a-priori (as well for a potential attacker).

Detection Latency provides a mechanism, how often the described process of the secure membership service should be executed. On the one hand the detection latency of one or more faulty nodes is adjustable. On the other hand computation and communication overhead is controllable.

We have implemented two versions of the membership service regarding the algorithm that is used for establishing security: one version is utilizing symmetric-key encryption only (AES) and the other one is using symmetric-key and public-key encryption combined (AES and RSA). As to the latter one, slaves use AES for encrypting global time and RSA is used to sign the membership vector on the master node.

3.3. AES

To improve security of the membership algorithm, the timing information will only be sent on the bus in encrypted form. This prevents the possibility of manipulating the signal by the intruder ($\rightarrow$ *integrity* aspect of security) to the value that is expected by the master, though the global time view of the slave node possibly differs and is compromised.

The *Advanced Encryption Standard* (AES) is a symmetric-key encryption algorithm developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen. The

name *symmetric* results from the fact that the same key is used for the encrypting and decrypting entity. Hence, the security of such a cipher depends only on the key which should be only known to two parties, so those ciphers are also sometimes called *private-key algorithms*. Symmetric ciphers can be divided into two categories: *block ciphers* and *stream ciphers*, whereas AES belongs to the former [KW06]. Block ciphers encrypt the messages in data blocks of fixed length (e.g. 64 bit), stream ciphers, in contrast are based on the *one-time pad* (OTP) encryption and are not discussed further in this thesis – see [KW06] for a short introduction.

AES emerged in the search of a replacement for the obsolete *Data Encryption Standard* (DES). DES became vulnerable to brute-force attacks in the end of the 90s due to its short key-size of only 56 bits.[5] In 2001, after a 5-year standardization process organized by the *National Institute of Standards and Technology* (NIST), in which fifteen competing designs were presented and evaluated, **Rijndael** was announced to be the winner because it was voted as the most suitable, that is to say, it had the best combination of security, performance, efficiency, implementability and flexibility, at the conferences, and thus became the new AES standard. It became effective as a Federal government standard in May 2002 and is now used worldwide[DR02].

AES has a fixed 128-bit block size and allows key sizes of 128, 192 or 256 bits, respectively. The only difference between Rijndael and AES is the fact that Rijndael supports more block sizes than just one (from 128 to 256 bits) and the key size has theoretically no maximum, but both need to be multiples of 32 bit.

AES consists of four invertible transformations which provide confusion and diffusion:

- Byte Substitution
- Shift Rows
- Mix Columns
- Add Round Key

Every step will described shortly on the basis of AES-128 to get an idea how Rijndael works. The basic block needed in this cipher is the so-called *State*, which is a 4 by 4 matrix of bytes, containing the arranged data of the actual 128-bit data block that needs to be encrypted. The keys are similarly arranged in a matrix of 4 by $\frac{\text{key length}}{32}$, so in our case also 4 by 4. The transformations are performed in several rounds, the details of those rounds and the exact order can be found in [DR02].

The first step describes a nonlinear, invertible **Byte Substitution** which is performed using a so called S-Box (which is in fact a fixed 256 byte array, constructed from mathematical considerations): for each byte in the State block an equivalent from the S-Box is chosen. This operation conforms to a substitution cipher. As the name indicates, the **Shift Rows** operation cyclically shifts each row of the two-dimensional State by a

[5] *Triple DES* (3DES), first published 1998, was an approach to increase the key size of DES by simply applying the original algorithm three times on each block. It is for example used in the electronic payment industry.

certain offset. The first row remains unchanged, the second row is shifted by one, the third row by two and the fourth row by three bytes to the left; the bytes are just getting permutated. Finally, the columns are mixed up (**Mix Columns**): every row of a column is muliplicated with a constant and the result is combined with a XOR operation. There is a complex mathematical theory about finite fields behind this procedure which builds, along with the shift-rows step, the primary source of diffusion in Rijndael. The **Add Round Key** function is the only one in AES which makes the algorithm dependent on the user key. Each byte of the State is combined with the round subkey using the XOR operation.

For our secure membership service implementation, a key size of 128 bits should be completely sufficient, however we also experimented with the other two larger sizes of 192 and 256 bits and compared the performance results (see Section 4.5.1 for the details).

3.4. RSA

RSA is an algorithm for public-key cryptography and is named after its inventors Rivest, Sharmir and Adleman. It is an asymmetric procedure, which means that there are two different keys: a public and a private key. The latter is used to encrypt data and therefore should be kept secret. The public key can be shared; with it one is able to decrypt a message signed by the corresponding private key. RSA is a deterministic algorithm, which means that every key-pair is theoretically breakable if enough computation power and enough time is available[6]. Thus, a longer key is preferred, but a longer key takes more computation time and memory usage on encryption respectively decryption, which are both quite limited in embedded systems.

The following example should illustrate how the idea behind RSA works:

1. choose two prime numbers: $p = 101$ and $q = 113$
2. calculate n:
$$n = p \cdot q = 11413$$
3. calculate $\varphi(n)$:
$$\varphi(n) = (p-1) \cdot (q-1) = 100 \cdot 112 = 11200$$
4. choose e, which must be coprime to $\varphi(n)$, under the requirement $e < \varphi(n)$. Without any particular reason, we choose $e = 3533$.
5. now we have to calculate d, which must fulfil following condition:
$$d \cdot e \equiv 1 \pmod{\varphi(n)}$$
and results into...
$$d = 6597$$

[6] it is basically a prime factorization problem, which is NP-hard

6. therefore, the public key is ($n = 11413$, $e = 3533$) and the private key ($n = 11413$, $d = 6597$).

7. to encrypt a message m, this equation must be solved (where c is the encrypted message and m is plain text), $m = 65$:

$$\begin{aligned} c &= m^d \pmod{n} \\ c &= 65^{6597} \pmod{11413} \\ c &= 5269 \end{aligned}$$

8. for decryption we take the public key pair:

$$\begin{aligned} m &= c^e \pmod{n} \\ m &= 5269^{3533} \pmod{11413} \\ m &= 65 \end{aligned}$$

Thus the principal idea is, that it is a way more easier to multiply two numbers than finding the factors of the result itself. For a more detailed explanation on RSA take a look at [Sch96].

For our secure membership service implementation, RSA matches in several requirements to encrypt the membership vector:

- **key management**: only the master must own the private key; the public key can be shared safely with the slaves. In contrast to symmetric algorithm, a compromised slave is not able to (theoretically) sign the faked membership vector because it does not own the private key.

- The problem with symmetric algorithm could be solved by using one key for each slave, but this would make key management harder as well and require more **memory**.

Using RSA naturally requires additional program memory for the implementation, since we use AES for encrypting the global time anyway.

4. Implementation

4.1. Hardware Setup

As hardware setup for the implementation of the FlexRay driver and the secure membership service we have used development boards by Fujitsu, called *bits pot blue*. "bits pot" denotes a series of five color-boards microcontroller starter kits [Mic10b] designed especially for learning automotive network technologies. The blue board is designed for FlexRay communication, other colored boards contain chips for CAN, USB and LIN communication.

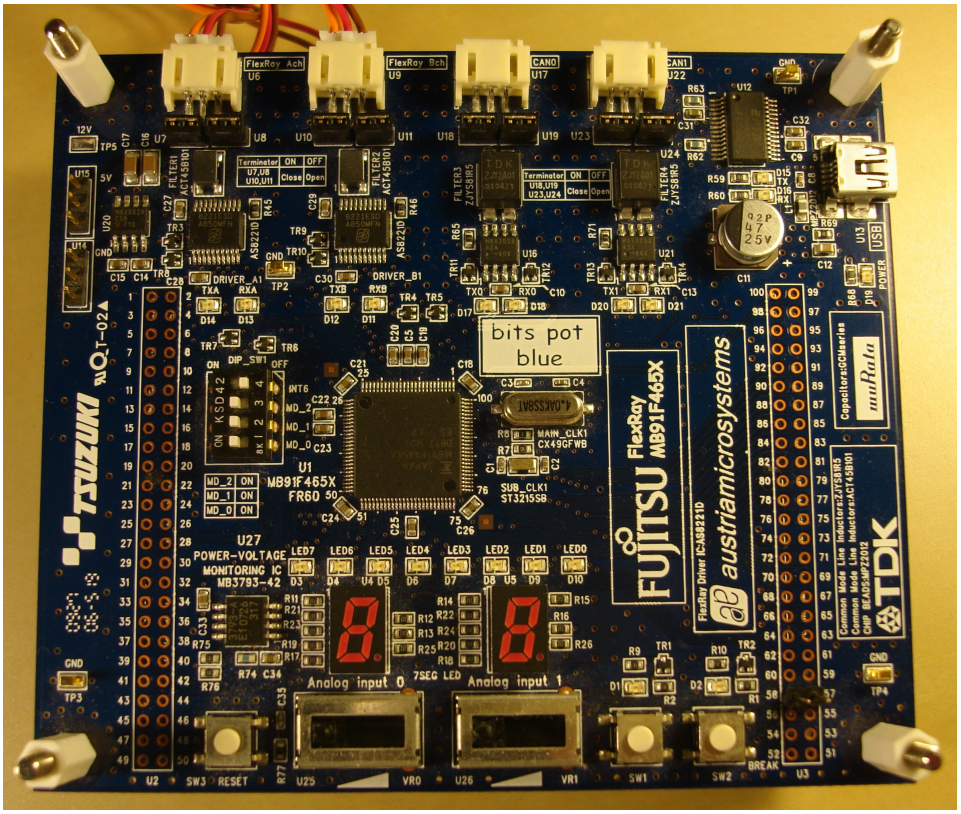


Figure 10: The "bitspot blue" Development Board

As you can see in Figure 10, the development board contains eight LEDs, two seven-segment displays, two volume switches (connected to the ADC of the microcontroller), two push buttons, one reset button and four connection ports, representing communication channel A and B of the FlexRay controller and two channels for CAN communication. A connection to the PC can be established via USB.

The microcontroller model included in this board is named **FR60 MB91F465X**[7].

[7] *FR* does not stand for FlexRay, but is the abbreviation of "FUJITSU RISC" controller, a product line of Fujitsu Microelectronics Ltd.

It contains the basic functions of the *FR60 MB91460* series, a "line of general-purpose 32-bit RISC microcontrollers designed for embedded control applications which require high-speed real-time processing such as consumer devices and on-board vehicle systems", plus an FlexRay IP-module (called *E-Ray*) that performs communication according to the FlexRay Specification [Con05]. The most important capabilities of the microcontroller are shown in Table 1.

CPU clock frequency (max.)	100 MHz
FlexRay clock frequency (max.)	80 MHz
Flash memory	544 KiB
D-RAM (Data)	16 KiB
ID-RAM (Instruction Data)	16 KiB
Flash Cache (Instruction cache)	8 KiB

Table 1: FR60 MB91F465X Properties

Figure 11 on page 29 shows a block diagram of the E-Ray IP-module and is described in detail in Section 4.3.2.

4.2. Toolchain

4.2.1. Softune™ Tools and the Port to Linux

The bitspot blue software package included a bunch of Windows tools to get started with the FlexRay board. The most important for development is the *Softune™ IDE*. However, since we use Linux, we have looked for a way to use the toolchain without being dependent on one single graphical tool (which is unintuitive to use – and was probably designed for Japanese users). It would be much more flexible to have the basic tools like compiler, assembler and linker on the command-line and call them from one's favourite editor or IDE, or – to save even more effort – put them in a comfortable Makefile (see next Section 4.2.2). By using *wine*, a well-known application that allows Windows programs to run on Unix-like operating systems, we can use the toolchain for this microcontroller under Linux.

4.2.2. Makefile

`make` [Fou10] is a well-known and popular software development utility, widely used on Unix-based platforms. Because we are used to `make`, we have decided to use it for our new environment too. Therefore, one of the central parts of the Linux port is the Makefile. It provides an easy-to-use command-line interface to the toolchain and simplifies the build-process by specifying some special targets – the most important being:

- `$ make` → the simplest call without arguments just builds the first node
- `$ N=2 make` → builds the second node - with the environment variable `N` a specific target node can be selected

- `$ make allnodes` → builds all nodes (there are two nodes by default; to change that, you have to modify the `FLASH_NODES` environment variable in the Makefile)
- `$ N=2 make upload` → flashes the binary of the second node to the board (again, by omitting the environment variable `N` the standard node 1 is selected); see section 4.2.3 for an explanation of the flashing tool used
- `$ make allupload` → flashes all node binaries to the boards, whereby node number n code is transferred to the device `/dev/ttyUSB`$[n-1]$

The building process can be adapted, tweaked or optimized by passing command-line flags to the tools involved (compiler, assembler and linker) – they can be modified in the Makefile in the variables `CFLAGS`, `ASFLAGS` and `LDFLAGS`, respectively. Note that some of the flags are already well-adapted to the bitspot blue target and should not be changed (for example, the memory areas of the different segments in the linker).

The trickiest parts of the Makefile are

- conversion between the different newline-perceptions [Wik10] by Windows and Unix
- proper dependency generation for header-files, to make it compatible with *make*
- adding configuration values to the registry of *wine*, to ensure proper functionality of the tools (e.g. finding standard headers of their library)

4.2.3. The flashing Tool `pyfrprog`

The given MB91F465X flashing tools for Windows were not very satisfying, and it was of course not very comfortable to switch to the Virtual Machine[8] all the time only for programming the boards, so we have decided to create our own tool running on Linux.

`pyfrprog` [UF10] is written in Python using the *Universal Serial Port Python Library* (USPP) by Isaac Barona Martinez [Mar06]. It gets a `.mhx`[9] file as input (output of the Softune™ linker/`f2ms` converter) and then just transfers its content to the board byte for byte. In order to start the transfer the RESET button needs to be pressed – this is because a communication with the internal BootROM application can only be established within the first 100ms after a reset. This application (which itself is not capable of any flash memory access) allows us to write a simple self-written kernel (called *pkernel*) into the board's instruction RAM. Afterwards, the BootROM is told to execute a certain address in RAM, which in fact starts the `pkernel`. The `pkernel` waits for commands on the UART and writes or rather reads data from flash memory. When the write and read operations are done, `pyfrprog` issues an operation-complete-command and therefore `pkernel` causes a restart of the microcontroller which boots the programmed application in the end.

[8] the flashing tools would not simply run under `wine` because there is no USB support so far; in `VirtualBox`, this works without problems

[9] The `mhx`-format is basically just a hexdump of the binary code plus some checksum information – it is very similar to the Intel HEX-Format

4.3. Used Libraries

To implement a secure membership service, different libraries are required. First, we need a library which provides common encryption operations such as AES and RSA. Second, we need a driver for the E-Ray IP-Core on the bits pot blue boards, since no driver is (publicly) available, except a demo application by Fujitsu with restricted functionality. In the following we describe the libraries we used and explain how we solved problems that occurred.

4.3.1. MatrixSSL

During the search for a encryption library we found *MatrixSSL*, which is an "embedded SSL and TLS implementation designed for small footprint application and devices" [Net10]. Therefore it perfectly fits in our usage of a lightweight embedded application. The architecture of the microcontroller however is not widely in use (at least in the FOSS[10]-field), so we had to port the library. Fortunately, the library is written very generic. As a result we basically had to set some constants through `#define` (e.g. byte order) and map some functions for our board (e.g. replace `printf()` with a function that uses the UART instead of `stdout`) which was quite an easy task. MatrixSSL also requires support for dynamic memory allocation for big numbers, which is in general provided by the C standard library call `malloc()`. The pitfalls we ran into are described in the next section. We also had to resize the stack size from 400 bytes to 800 bytes.

`malloc()` and friends

A more confusing step was the usage of `malloc()`: Although Fujitsu already provides a functional implementation of `malloc()` itself, it is not fully operative per se – something we found out the hard way: `malloc()` is available as usual with a simple

```
#include <stdlib.h>
```

and linking it against `stdlib`. As the tools do not output any warning or the like, our assumption was that everything would be fine. After *hours* of debugging, why MatrixSSL was not working correctly, we finally looked at `malloc()` again, and found out that `malloc()` always returns the same value, namely a `NULL`-pointer, which is obviously pretty bad.

After several hours, we found a hint in a manual [Mic08], how dynamic allocation can be added for their simulator in the Softune™ IDE:

```
#define HEEP_SIZE 16*1024
static long brk_siz = 0;

typedef int _heep_t;
#define ROUNDUP(s) (((s)+sizeof(_heep_t)-1)&~(sizeof(_heep_t)-1))
```

[10] Free and Open Source Software

```
static _heep_t _heep[ROUNDUP(HEEP_SIZE)/sizeof(_heep_t)];
#define _heep_size ROUNDUP(HEEP_SIZE)

extern char *sbrk(int size)
{
    if(brk_siz + size > _heep_size || brk_siz + size < 0)
        return ((char *)-1);
    brk_siz += size;
    return ((char *)_heep + brk_siz - size);
}
```

In fact, we have to implement the low level function `sbrk()` and pack it into a library with the existing library of Fujitsu `stdlib` (which is, by the way, only supplied as compiled library). After we had done that, everything worked pretty nice.

The troubles however were not for nothing: Since we are able to control memory allocation on such a low level, we can easily determine dynamic memory usage by accessing the variable `brk_siz` (compare Section 4.4.4).

4.3.2. FlexRay Stack for the E-Ray IP-Module

As no E-Ray compatible driver was available, we had to write our own FlexRay implementation for that. This actually happened within a "practical training project" at university. Basically we tried to be compatible with [AUT09]. In this section we want to describe how the communication controller works and particularly how receiving and transmitting of messages are realized with the E-Ray IP-module.

In Figure 11 the block diagram of the E-Ray IP-module is depicted. In our case

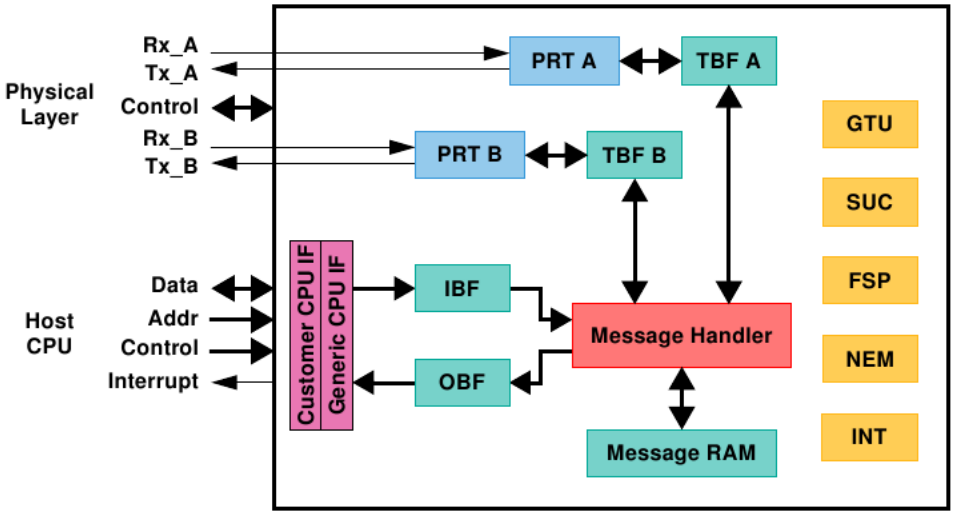


Figure 11: Schematic of the E-Ray IP-Module [Bos06]

the *Host CPU* is the FR60 MB91F465X which is connected with some data, address, control and interrupt lines of the E-Ray IP-module. Even some registers are mapped to the Host CPU (not pictured in the figure). For physical connection to other nodes two channels are available (`Rx_A & Tx_A` and `Rx_B & Tx_B`). The *Message RAM* holds several information about messages (e.g. Frame-ID, should the message issue a interrupt after arrival and so forth) and data of these messages. The Host CPU can not directly access the Message RAM; it has to request a specific message through the *Input Buffer* (IBF) and the *Output Buffer* (OBF), respectively.

Message RAM

The Message RAM is organized in $2048 \cdot 32$-bit words, where each word is protected with an additional parity bit. The organization of the content is illustrated in Figure 12.

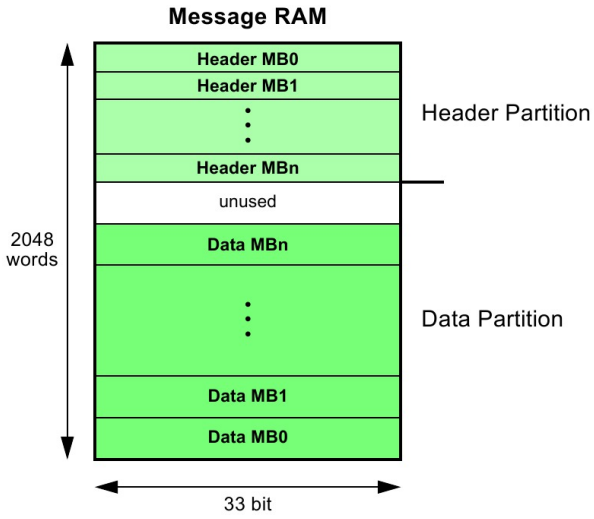


Figure 12: Illustration of the Message RAM [Bos06]

The Message RAM is basically split up into a region for headers and another one for data of messages. A header contains several information about a specific message as depicted in Figure 13 and has a size of $4 \cdot 32$-Bit. Depending on the configuration, the Message RAM can contain up to 128 messages and each data section can hold up to 254 bytes, which is the maximum size for the payload according to the FlexRay specification [Con05].

The configuration of the headers usually happens at the initialization of the communication controller, but can also be updated online. An essential configuration field is the *Frame-ID*: It specifies on which position a message is sent in the communication cycle. The *Data Pointer* refers to the position in the Message RAM where the relevant data for this message reside; the length is determined by the entry *Payload Length*. Thus,

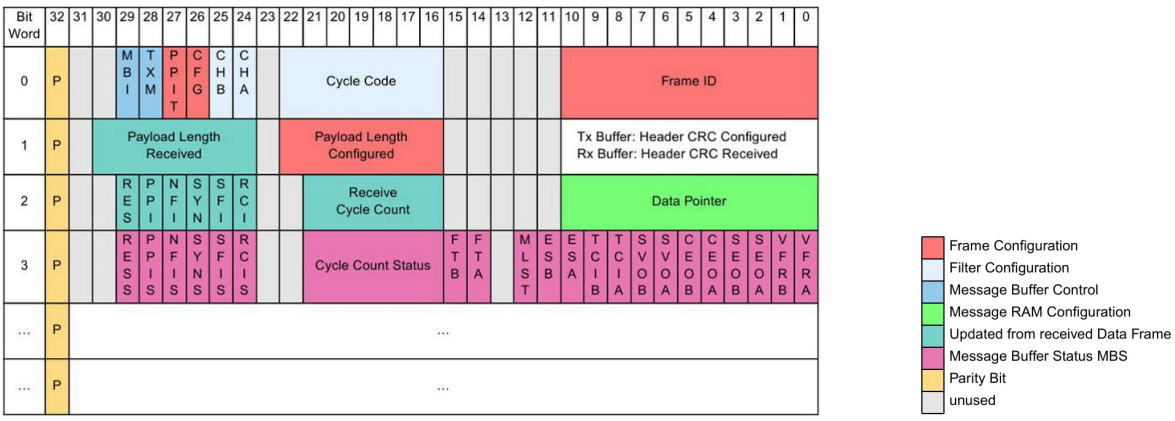


Figure 13: Illustration of the Message Header [Bos06]

it must take care while designing the layout for the Message RAM, because the E-Ray IP-module does not check whether data sections of different messages overlap in the Message RAM. The fourth word contains status information. The bit **VFRA** for example denotes whether a valid frame was received on channel A. For a full explanation of all fields see [Bos06] section 5.12.

Output Buffer (OBF)

As mentioned above, the Host CPU can not directly access the internal Message RAM of the E-Ray IP-module. Therefore, to receive the data of a configured message, a specific message must be requested from the Host CPU, which is loaded into the shadow buffer of the OBF (pictured in Figure 14). Once the message is in there, the Host CPU can issue a swap-command and the desired message can be read by the Host CPU from the host buffer through registers mapped to it. In detail, the following steps are necessary for a successful receiving of a message:

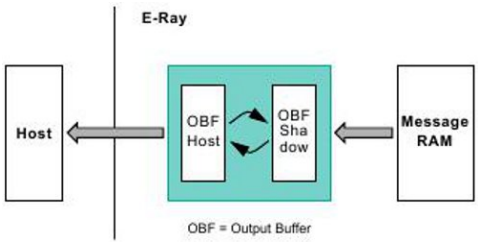


Figure 14: The Output Buffer [Bos06]

1. any action that triggers this process, e.g. an interrupt by the communication controller which indicates that a new message has arrived
2. set the message number[11] and whether header and/or data section should be loaded into the shadow buffer
3. start the transfer and wait until it is done
4. swap the shadow buffer with the host buffer
5. read the desired data from the registers

Input Buffer (IBF)

The IBF is provided for transmitting messages. It has a similar structure as the OBF, as depicted in Figure 15. Again, we describe the steps which are required for sending a message (or, to be more precise, to set the data of a message, because the communication controller itself decides when a message is sent):

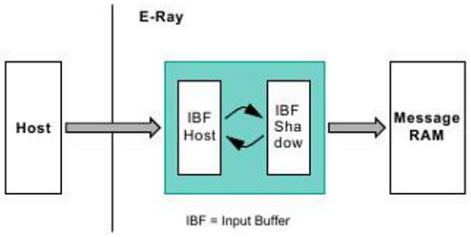


Figure 15: The Input Buffer [Bos06]

1. check if a transfer is already in progress and if so, wait until it is finished
2. load data into the host buffer
3. set command bits like the header configuration should be overwritten
4. set the message number of the desired message; this action swaps host and shadow buffer automatically

[11] which is not necessarily equal to the Frame-ID

Note that state and event messages are handled exactly in the same manner from a programmer's point of view. The configuration of the Message RAM, is further separated into a state and event message segment, where the border can be defined (e.g. totally 12 messages → 8 state and 4 event messages reside in the Message RAM in a row).

4.4. Implementation Details

4.4.1. Secure Membership Service

Global Time and FlexRay

According to [Con05, p. 170] FlexRay's global time consists of a *cycle counter* and a *macrotick value*. This is also true for the E-Ray IP-Module by Bosch which fulfils the official FlexRay Specification [Bos06]. Valid values for the cycle counter are in the range from 0 to 63. Since a period of a communication cycle with our configuration (cf. Section 4.4.2) is only 8.8ms long, the time horizon is about half a second which is quite few compared to other time-triggered protocols like TTP and TT-Ethernet. If we would have used the cycle counter, our implementation would be highly vulnerable to replay attacks.

Consequently we have enhanced the provided global time within the membership implementation, by increasing a 32bit counter on each cycle counter overflow – we call this *extended global time*. In our configuration of the FlexRay setup, it overflows after

$$2^{32} \times 64 \times 8.8\text{ms} \sim 76 \text{ years}$$

therefore it is surely sufficient for our purpose.

In order to increase the security aspect even more, the starting value upon a hardware reset of the counter is initialized with a random value generated by the noise of an unconnected ADC. For detailed source code compare Appendix B.4 and B.1.

Settings in `membership.h` (Appendix A.1)

The main configuration part is done by setting several constants in the header file `membership.h`. The important ones are explained in the following:

- `MAX_SLAVES`: maximum number of slaves that can be managed by the implementation.
- `NO_SLAVES`: number of slaves that are actually in use (must be lower than `MAX_SLAVES`).
- `FRAMEID_SLAVEX`: describes at which Frame-ID a slave X sends its encrypted global time information.
- `FRAMEID_EXTGT`: describes the Frame-ID which contains the extended global time information.
- `FRAMEID_MBVEC`: describes the Frame-ID to use by the master to send the encrypted membership vector.

- `FAULTY_SLAVE`: by setting this constant, one can simulate a manipulated slave which sends wrong timing information to the master and therefore is excluded by the master and at the resulting membership vector respectively.
- `FREQUENCY`: defines how often the membership process is executed (compare Section 4.5).
- `RSABITSIZE`: declares the RSA bitsize which should be used; must be divisible by 8.
- `AESBITSIZE`: declares the AES bitsize which should be used; must be 128, 192 or 256.

Functions of the Secure Membership Service

The following list shortly describes the most important functions our secure membership service implementation consists of. Each procedure is either designed for being called on the master node (marked with → [**M**]) or on one of the slaves (→ [**S**]). For a better understanding the functions are listed in the order of their logical steps as described in Section 3.2. The full source code of the implementation resides in the files `membership_master.c` respectively `membership_slave.c` and can be found in Appendix B.1 and B.2.

```
void SecMem_Init_Master(void);
```
[**M**]

This function starts the secure membership service for the master node. Cryptography routines for AES and RSA are initialized and necessary interrupt handlers are set. Additionally, the function `SecMem_SendMembershipVector()` which is described below is added as a task to the scheduler. From a programmer's perspective, this initialization function is the only one that needs to be called on the master for a fully working secure clock synchronization; other important functions are automatically called via interrupts and the scheduler.

```
void SecMem_Init_Slave(void);
```
[**S**]

This is the counterpart of the previous initialization function for the slaves. Again, no other function needs to be called by the programmer, as it already registers the necessary interrupt handler (for `SecMem_ReceiveMembershipVector()`) and the task for sending the global time to the master periodically (see `SecMem_SendTimeToMaster()`).

```
void SecMem_SendTimeToMaster (void);
```
[**S**]

This function represents the first logical step of the secure membership service: the slave's own view of the global time is determined via a call to the FlexRay driver (→ `Flxr_GetGlobalTime()` in combination with the extended global time) and then symmetrically encrypted with AES. The result is then sent to the master. Note that a message block is longer than the global time value (which is in fact only one byte for the cycle counter and four bytes for our global time extension), so the rest of the block is filled up with random values before the encryption takes place for security reasons.

The function is not thought to be called by hand, but periodically by the scheduler at the start of each communication cycle.

`void SecMem_ReceiveSlaveTime(uint8 *encr, uint8 len, sint8 slno);` [M]
The next logical step of our implementation requires the receiving of the encrypted time information, constructed by the slave via `SecMem_SendTimeToMaster()`. After encrypting the message block, the received global time is compared with the master's own view of global time; if the values differ, the slave is considered to be faulty and the corresponding entry in the membership vector (indicated by the `slno` parameter) is set to zero.

`void SecMem_SendMembershipVector (void);` [M]
In the next step, after all slaves have sent their global time view to the master and the membership vector information is fully constructed, the master has to send the vector to all slaves to notify them whether they are perhaps out of membership respectively whether they can still trust certain other nodes or not. This transmission of the vector happens in the listed function. Like in `SecMem_SendTimeToMaster()`, the information is encrypted before – this time with the AES or RSA algorithm, depending on the chosen implementation. Again, because the vector data is likely smaller than the message block size, the rest is filled up randomly for security reasons before encryption. In addition the extended global time is sent unencrypted in an explicit slot to propagate the timing information.

`void SecMem_ReceiveMembershipVector(uint8 *encr, uint8 len);` [S]
In the end of the secure membership service process, the constructed membership vector has to be received from the master by all slave nodes. As it has been encrypted, this routine has to decrypt the message block. If the slave node notices by means of the membership vector that it is out of membership, it turns into a safe state.

`void SecMem_ReceiveExtGT(uint8 *data, uint8 len);` [S]
This function is a callback function which applies the extended global time information received by the master to a local counter of the slave.

`void SecMem_ShowMembershipVector(uint8 memvector);` [M,S]
This function only exists for debugging purposes and sends a human readable representation of the membership vector `memvector` to the UART interface via `dprintf()`.

4.4.2. Demo Application

In order to test our secure membership implementation we had to develop a lightweight application, which uses basic functionality of the FlexRay driver using the provided interfaces of the evaluation board. As labelled in Figure 16 each device has the following role:

A1, A2 On the left seven-segment display a simple counter prints its output, which is

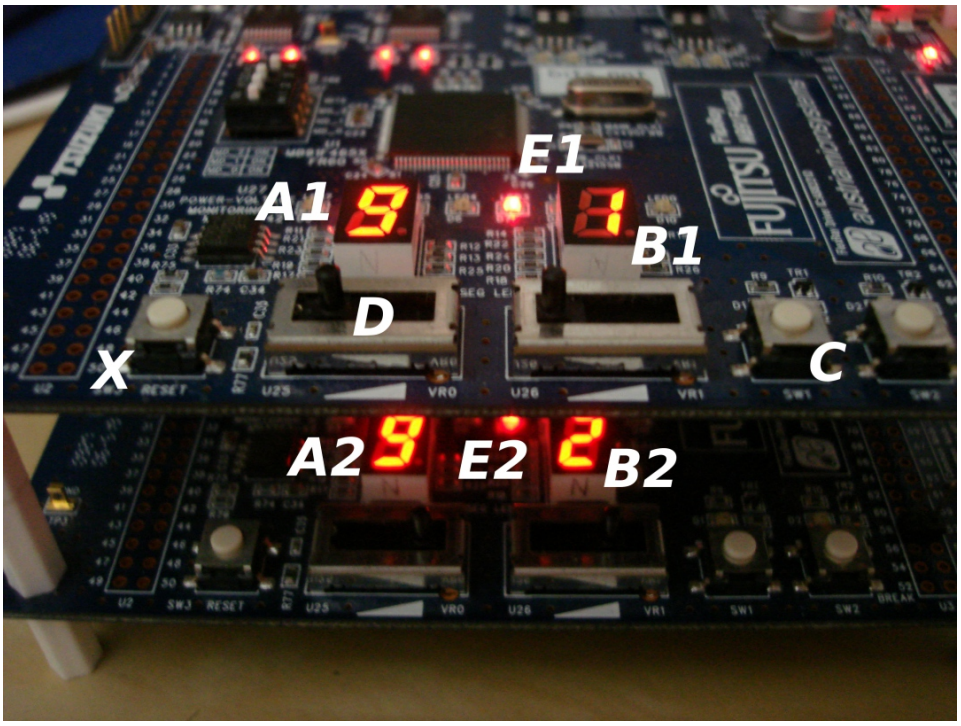


Figure 16: Explanation of the used I/O in the Application

controlled by the master: A so-called *reloadtimer* on the master node is configured, which generates an overflow interrupt at approximately every second. In the interrupt service routine a counter is incremented by one and an event message with the payload "1" is sent afterwards. By receiving the message on the slave, its local counter is increased by one as well. It counts from 0 to F. On an overflow it starts at 0 again.

This should demonstrate the use of event messages, although it would be more appropriate to use static messages here. Also note, that the boards have to start counting at the same time, otherwise the local counters will differ to each other (even if they count up at the same time[12]) which would not be the case for static messages.

B1, B2 The right seven-segment devices display the corresponding node ID. In this case, node ID 1 is the master of the membership protocol and node ID 2 is the slave.

D, E1, E2 The volume control is internally connected to an ADC[13]. The converted result is displayed on four LEDs (E1)[14] and is also sent in the FlexRay slot 1 of the communication cycle as a static message. On receiving at the slave, the value is outputted on its LEDs (E2) too. In practice this happens immediately, in the user's point of view.

[12] to be correct: *nearly* at the same time – this however is not visible to the eye
[13] Analog-to-digital converter
[14] which are better depicted in Figure 10

C By pressing this button on the master node, an additional feature is turned on: instead of just displaying the ADC value, it will blink synchronously with approximately 1 Hz on both nodes. This is easily achievable by using static messages. By pushing it again the original behavior will be restored.

X This button resets the MCU.

(a) FlexRay Settings

Property	Value
baud rate on FlexRay bus	10 MBit/s
one microtick	25 ns
one macrotick	1 μs
communication cycletime	352000 microticks
communication cycletime	8800 macroticks
communication cycletime	8,8 ms
payload static segment	254 bytes
number of static slots	24 slots
static slots length	272 μs
static segment	6.53 ms
payload dynamic segment	0–254 bytes
number of minislots	308 slots
minislot length	7 μs
dynamic segment	2.12 ms

(b) Slot Roadmap

FID	from	Type	Description
1	[**M**]	state	LED value from ADC
4	[**S1**]	state	encrypted global time
6	[**S2**]	state	encrypted global time (reserved)
7	[**M**]	state	extended global time information
8	[**S3**]	state	encrypted global time (reserved)
9	[**M**]	state	membership vector
–	[**M**]	event	payload for counter @ 7seg

Table 2: FlexRay Configuration of the Application

A further important property of our application is the specific FlexRay configuration. Details of it are listed in Table 2.

4.4.3. Simple Task Dispatcher

In order to execute simple tasks, we had to implement a task dispatcher. Basically, this happens with a timer interrupt, e.g. after a timer overflow a certain task set should be executed. However, a requirement was to start this task set at each start of a communication cycle. Principally, we would have been able to achieve this requirement by using a local timer and syncing it by a special message in the communication cycle, but there is a better solution: Fortunately, the E-Ray IP-module provides an own interrupt for that, called *Cycle Start Interrupt* or *Begin of Cycle (BOF) Interrupt*.

In order to determine the current cycle, the function `GetGlobalTime()` can be used, which returns a pair consisting of the cycle- and the macroticks-counter, as has already been stated in Section 3.2. This fact is vital for a dispatcher implementation, because BOF interrupts can get lost due to different circumstances, particularly for a short cycletime. On a high processor load a CPU-intensive taskset for example can not finish

until the next cycle starts and the flag can not be cleared in time. Therefore we have to manage the latest cycle value to calculate how many interrupts we have missed. This is not particularly that helpful when we want to execute the taskset at every cycle-start, therefore we introduced a way to start the execution at every xth cycle, where x is defined by `DISPATCHER_PRESCALE`. This functionality is fulfiled by the private function `dispatcher_newcycle()`.

A task can be added by calling `dispatcher_addTask()` which requires three parameters:

index identifies a specific task and defines the position in the taskset.

freq means, that the given task is executed every `freq`th dispatcher cycle[15].

handler a callback function which should be called on the appropriate instant.

Analogously a function `dispatcher_removeTaks()` is provided for removing tasks.

Last but not least, the function `dispatcher_loop()` must be placed in the infinite loop of `main()`. `dispatcher_loop()` executes all registered task and meets the given `freq` conditions. For the full sourcecode listings, compare Appendix A.2 and B.4.

4.4.4. Evaluating Performance

In this section we want to outline how we measured processor and memory utilization on the Fujitsu evaluation boards. On the running application, the calculated results are printed periodically on the UART device. A detailed summary of the results are provided in the next Section 4.5.2.

Processor Utilization

The goal is to compute the system load over a period of time. We have implemented a counter which is incremented by one in the `main()`-loop. Therefore, the counter is incremented every time the processor is actually inactive[16], otherwise the CPU is busy with an interrupt service routine or a task of the dispatcher. Thus, the greater the value of the counter, the smaller the processor utilization is. To obtain a relative number in percent, a reference tickvalue has to be determined; this value is simply gained by disabling all tasks and interrupts[17] (compare `TICKS_PER_ROUND_CLEAN` in Appendix A.3).

By using a reloadtimer running at 100Hz, we can determine the processor load for a certain period. Furthermore, there is a constant `PERF_ROUNDS` to compute the average load over a given amount of rounds. Maybe this approach is not the most exact one, but it is absolutely sufficient for our purpose, making a comparison between different settings of the secure membership implementation.

[15] this is not necessarily equal to a communication cycle (consider the configoption `DISPATCHER_PRESCALE`)

[16] we did not use any sleep mode for our application

[17] except the UART device of course

Memory Usage

Two values are important for memory: static and dynamic memory usage. Static memory usage can be simply determined after compilation and linking of the program by using the given tools of Fujitsu. In fact, the linker generates a detailed report of the static memory layout at compile-time.

The dynamic memory is determined by our `sbrk()`-implementation as mentioned already in Section 4.3.1. Since we have to implement `sbrk()` anyway, we can easily access those values which are relevant for calculation of the usage.
For full sourcecode listings see Appendix A.3 and B.5.

4.5. Analysis

In this section we have inspected our implementation of a secure membership regarding processor load and the usage of memory at compile-time (static) and at run-time (dynamic). We have compared different settings, which means we use different key sizes and modify the frequency which has an influence on when a membership vector should be processed.

Keylength for Encryption

AES For AES we selected all available key sizes, namely 128, 192 and 256 bits.

AES-RSA Since AES256 is not that more expensive than AES128 (this is true for MatrixSSL at least, cf. Section 4.5.1), AES256 is used for encrypting the global time.

Encryption of the membership vector uses RSA. We have selected the following key sizes to analyse: 128, 256, 512, 1024 and 1536 bits.

We have also included measured data with disabled membership service, to declare a lower bound.

Frequency

In order to minimize CPU load, we have implemented a way to determine how often the global time is sent or the membership vector is received[18] respectively. We call that *frequency*. However, there are two approaches to realize this idea, which are not mutually exclusive:

- `DISPATCHER_PRESCALE (DP)`: specifies how often the taskset of the dispatcher should be executed, whereby this constant is a division factor: the execution of tasks happens every `DP`th time.
- `FREQUENCY (FQ)`: this constant is similar to `DP`, thus it does the same for a single task. In fact, our dispatcher introduced in Section 4.4.3 provides such a mechanism.

[18] since the membership vector is sent as a static message, this is no restriction

Although the functionality is similar, those parameters depend on each other, e.g. DP=10 and FQ=10 result in an overall-frequency of 100, meaning that the membership task is executed every 100th communication cycle of the FlexRay controller, because the dispatcher relies on the `begin of cycle`-interrupt as discussed in Section 4.4.3. In order to simplify the results only the overall-frequencies are provided in the results. For detailed implementation see Appendix B.1, B.2 and B.4.

4.5.1. AES

Processor Utilization

In Table 3 all results are printed. Since the results are not significant enough, we can conclude that it does not make a difference which configuration is chosen. Therefore AES256 is recommended.

Configuration	Frequency Period	100 880.0 ms [%]	10 88.0 ms [%]	5 44.0 ms [%]	3 26.4 ms [%]	1 8.8 ms [%]
No Secure Membership	Master	65.4 %	65.4 %	65.4 %	65.4 %	65.4 %
	Slave	50.5 %	50.5 %	50.5 %	50.5 %	50.5 %
AES**128**	Master	65.5 %	66.6 %	65.7 %	65.9 %	67.7 %
	Slave	50.6 %	52.4 %	50.7 %	51.1 %	53.4 %
AES**192**	Master	66.8 %	66.8 %	65.7 %	66.0 %	67.7 %
	Slave	52.4 %	52.4 %	50.7 %	51.1 %	53.3 %
AES**256**	Master	66.8 %	65.5 %	66.9 %	67.3 %	66.4 %
	Slave	52.4 %	50.6 %	52.6 %	52.9 %	51.7 %

Table 3: CPU Utilization for different Configurations (AES)

Memory Usage

Static usage is with every configuration 2120 bytes on the master and 2104 bytes on the slave, this is about 20% of the available memory. Since MatrixSSL can not distinguish between the bit size at compile-time by design, the usage do not vary between the configurations. Dynamic memory is not required for the AES implementation.

4.5.2. AES-RSA

Processor Utilization

In Table 4 all results are included. The CPU utilization for the master is depicted graphically in Figure 17. Some results in Table 4 were not fully operative however, therefore they are signed with a symbol:

$\star$ in such cases, the dispatcher has not been able to complete all tasks within the given timing constraints. The consequences are that other tasks might be significantly delayed and that the encrypted membership vector is not broadcasted that often as it should be. This is also one reason why the CPU utilization does not distinctly

Configuration	Frequency Period	1000 8800.0 ms [%]	100 880.0 ms [%]	10 88.0 ms [%]	5 44.0 ms [%]
No Secure Membership	Master	65.4 %	65.4 %	65.4 %	65.4 %
	Slave	50.5 %	50.5 %	50.5 %	50.5 %
RSA**128** AES256	Master	65.4 %	66.2 %	70.1 %	74.8 %
	Slave	50.7 %	50.8 %	51.5 %	52.3 %
RSA**256** AES256	Master	65.8 %	68.6 %	78.6 %	⋆ 85.4 %
	Slave	50.7 %	50.9 %	51.7 %	52.3 %
RSA**512** AES256	Master	69.1 %	79.0 %	92.6 %	⋆ 96.0 %
	Slave	52.5 %	52.9 %	53.4 %	53.5 %
RSA**1024** AES256	Master	82.2 %	⋆ 95.7 %	⋆ 98.6 %	⊗ 99.6 %
	Slave	52.5 %	52.6 %	52.7 %	52.7 %
RSA**1536** AES256	Master	⋆ 100.0 %	⋆ 100.0 %	⋆ 100.0 %	⊗ 100.0 %
	Slave	52.5 %	52.5 %	52.6 %	52.6 %

Table 4: CPU Utilization for different Configurations (AES-RSA)

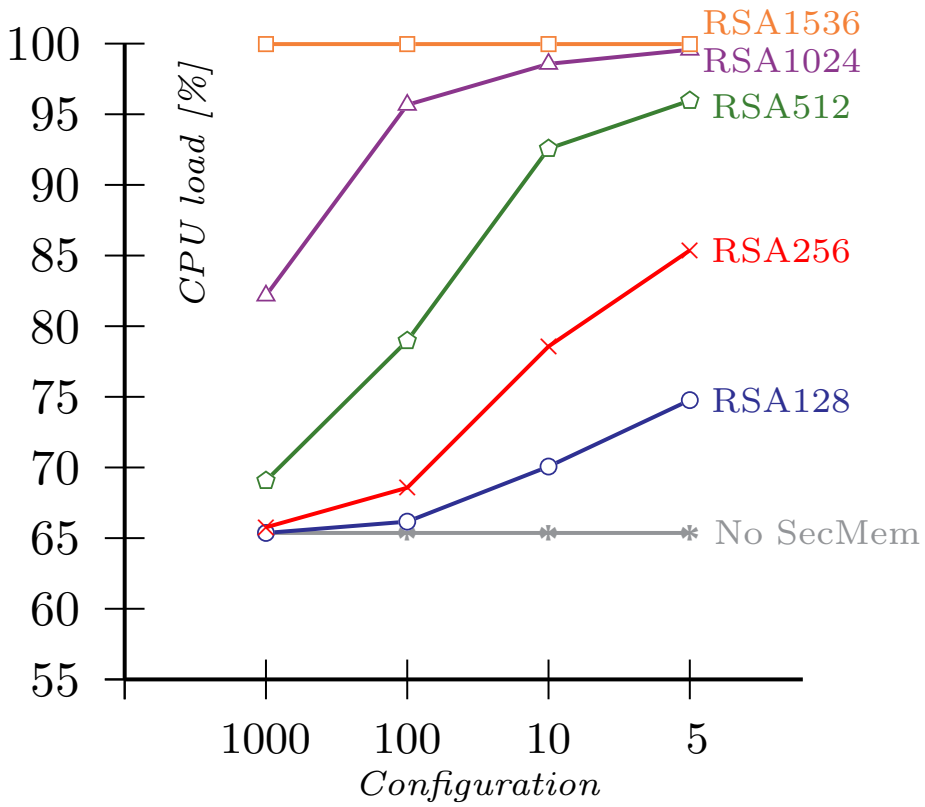


Figure 17: CPU Utilization on the Master (compare Table 4)

increase on the slave: Since the membership vector is not received that often, the decryption happens less frequently[19]. However, the application inclusive the secure membership service is still operating.

⊗ in those cases the timing requirements are so hard that even the membership service is not fully working.

Memory Usage

Configuration		Dyn. Mem. [*byte*]	Dyn. Mem. [%]	Static Mem. [*byte*]	Static Mem. [%]
No Secure	Master	0 *byte*	0.00 %	1056 *byte*	9.38 %
Membership	Slave	0 *byte*	0.00 %	1056 *byte*	9.38 %
RSA**128**	Master	640 *byte*	6.94 %	1740 *byte*	24.27 %
AES256	Slave	380 *byte*	4.12 %	1724 *byte*	24.05 %
RSA**256**	Master	1712 *byte*	18.58 %	1772 *byte*	24.72 %
AES256	Slave	636 *byte*	6.90 %	1740 *byte*	24.27 %
RSA**512**	Master	3024 *byte*	32.81 %	1836 *byte*	25.61 %
AES256	Slave	1148 *byte*	12.46 %	1772 *byte*	24.72 %
RSA**1024**	Master	5648 *byte*	61.28 %	1964 *byte*	27.40 %
AES256	Slave	2172 *byte*	23.57 %	1836 *byte*	25.61 %
RSA**1536**	Master	8272 *byte*	89.76 %	1992 *byte*	27.79 %
AES256	Slave	3196 *byte*	34.68 %	1900 *byte*	26.51 %

Table 5: Static and dynamic Memory Usage for different Configurations (AES-RSA)

While the frequency-settings influence CPU load, this consideration is not true for memory of course. Thus, we outsourced the data in a separate table (cf. Table 5). As shown in Table 1 on Page 26, the microcontroller includes a 16 KiB RAM-block. Due to the usage of `malloc()`, the memory is divided into a section for dynamic and static memory, whereby the former one has to be determined at runtime (as described in Section 4.4.4). In this particular case we have reserved 9 KiB of memory for dynamic usage. As can be seen, with RSA1536 just the *dynamic* memory uses *over* the half of the whole RAM on the master, which is remarkably much memory for a membership service.

[19] the second reason is that decryption does not take that long

5. Conclusion

With our model of a secure membership service, the system is able to *detect* whether a specific node is part of the membership vector or not. It prevents attacks like presented in our Attacker Model, e.g. delay attacks on a certain node. We have developed a reference implementation for the well-known time-triggered protocol FlexRay, although the secure membership service is not designed for a specific time-triggered platform. In fact it is practicable with some effort for other protocols like TTP and TT-Ethernet.

The overhead for our implementation was experimentally determined for several configurations – adjustable by encryption length and detection latency – with regard to CPU utilization and memory usage on the development boards of the "bitspot blue" series.

5.1. What Configuration should be used for my real World Application?

We cannot give an exact answer to this question. It extremely depends on your environment and the application you want to realize. Since resources are very limited in embedded systems, you may take into consideration that you do not even need a *secure* membership for your application anyway. Also keep in mind that RSA768 is already crackable [KAF+10] with some effort. Therefore a higher key size for a safety-critical application should be considered. According to [Kal03] a key size of 2048 bits should be sufficient until the year 2030. The MCU on the evaluation board for example is definitively overloaded with RSA1536, therefore – if really needed – a better MCU regarding CPU power and RAM should be considered.

5.2. Future Work

The primarily flaw of our secure membership service is that it is based on a master-slave system: A distributed approach as for example presented in [MLKSP10] seems to be a great enhancement towards the current idea. Further it would be interesting if an implementation on a different time-triggered platform like TTP and TT-Ethernet is also (easily) possible.

With regard to our specific implementation results for a different and faster hardware setup would be an interesting comparison. Also other libraries for RSA which are particularly adapted for embedded systems like in [GPW+04] or even hardware support realized with the help of FPGAs like mentioned in [WK06, p. 154] could be worthwhile enough for further investigations on this topic.

References

[AUT09] AUTOSAR. *Specification of FlexRay Driver, V2.3.0, R4.0 Rev 1*, 2009.

[BMR00] J. Berwanger, Peller M., and Griessbach R. byteflight - A New High-Performance Data Bus System for Safety-Related Applications. 2000. `http://www.byteflight.com/presentations/byteflight_paper.pdf`.

[Bos06] Bosch. *E-Ray – FlexRay IP-Module, User's Manual, Revision 1.2.5*, 2006.

[BPS08] J. Berwanger, M. Peteratzinger, and A. Schedl. Flexray startet durch – FlexRay-Bordnetz für Fahrdynamik und Fahrerassistenzsysteme. In Elektronik automotive: Sonderausgabe 7er BMW. 2008. `http://www.elektroniknet.de/home/automotive/bmw-7/flexray-startet-durch/` (unfortunately offline).

[Con05] FlexRay Consortium. *FlexRay Communications System Protocol Specification, Version 2.1, Revision A*, 2005.

[Dav10] Matt Davis. 2011 Audi A8 4.2 FSi First Drive, 2010. `http://www.insideline.com/audi/a8/2011/2011-audi-a8-4-2-fsi-first-drive.html`.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael – AES - The Advanced Encryption Standard*. Springer-Verlag, 2002.

[Fou10] Free Software Foundation. GNU Make, 2010. `http://www.gnu.org/software/make`.

[GPW+04] Nils Gura, Arun Patel, Arvinderpal W, Hans Eberle, and Sheueling Chang Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. pages 119–132, 2004.

[KAF+10] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Pierrick Gaudry, Peter L. Montgomery, Dag Arne Osvik, Herman Te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus, 2010.

[Kal03] Burt Kaliski. TWIRL and RSA Key Size, 2003. `http://www.rsa.com/rsalabs/node.asp?id=2004`.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[KW06] Sandeep Kumar and Thomas Wollinger. Fundamentals of symmetric cryptography. In Kerstin Lemke, Christof Paar, and Marko Wolf, editors, *Embedded Security in Cars – Securing Current and Future Automotive IT Applications*, pages 125–143. Springer-Verlag, 2006.

[Mar06] Isaac Barona Martinez. USPP – Universal Serial Port Python Library, 2006. `http://sites.google.com/site/ibarona/uspp`.

[Mic08] Fujitsu Microelectronics. *FR Family Softune^{TM} C/C++ Compiler Manual for V6*, 2008.

[Mic10a] Fujitsu Microelectronics, 2010. `http://www.fujitsu.com/global/services/microelectronics/technical/flexray/index_p13.html`.

[Mic10b] Fujitsu Microelectronics. bits bot – microcontroller starter kits by Fujitsu, 2010. `http://www.fujitsu.com/global/services/microelectronics/product/micom/tools/hard/board/bitspot.html`.

[MLKSP10] Martin Mitzlaff, Michael Lang, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. A Membership Service for a Distributed, Embedded System Based on a Time-Triggered FlexRay Network. In *Dependable Computing Conference (EDCC), 2010 European*, pages 155 –162, 28-30 2010.

[Net10] PeerSec Networks. PeerSec Networks MatrixSSL, 2010. `http://www.matrixssl.org`.

[Rau07] Mathias Rausch. *FlexRay. Grundlagen, Funktionsweise, Anwendung.* Hanser Fachbuch, 2007.

[RG91] Debby Russell and G.T Gangemi. *Computer Security Basics.* O'Reilly Media, 1991.

[Sch96] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition.* Wiley, 2nd edition, October 1996.

[UF10] Bernhard Urban and Sebastian Falbesoner. a native linux programmer for MB91F465X, 2010. `http://github.com/lewurm/frprog`.

[Wik10] Wikipedia. Newline — Wikipedia, the free encyclopedia, 2010. `https://secure.wikimedia.org/wikipedia/en/w/index.php?title=Newline&oldid=366510232#Common_problems`.

[WK06] Thomas Wollinger and Sandeep Kumar. Fundamentals of asymmetric cryptography. In Kerstin Lemke, Christof Paar, and Marko Wolf, editors, *Embedded Security in Cars – Securing Current and Future Automotive IT Applications*, pages 146–156. Springer-Verlag, 2006.

List of Figures

List of Tables

A. Header Files

A.1. membership.h

```
1  /*
2  Copyright (C) 2010   Bernhard Urban <lewurm@gmail.com>
3  Copyright (C) 2010   Sebastian Falbesoner <sebastian.falbesoner@gmail.com>
4  All rights reserved.
5  */
6
7  #ifndef __MEMBERSHIP_H
8  #define __MEMBERSHIP_H
9
10 #include "Platform_Types.h"
11 #include "..\defs\_fr.h"
12 #include "Flxr.h"
13 #include "dispatcher.h"
14 #include "uart.h"
15 #include "..\matrixssl\crypto\cryptoApi.h"
16 #include "..\keys.h"
17
18 /*************************************************************
19  * A Secure Membership Service for Time-Triggered Protocols *
20  *************************************************************/
21
22 #define MAX_SLAVES 4
23 #define NO_SLAVES 1 //number of slaves
24
25
26 #if NODE >= 2
27 #define SLAVE_NUMBER (NODE-2)
28 #endif
29
30
31 #define FRAMEID_SLAVE0 4 //Frame IDs have to be in a "row", otherwise
32 #define FRAMEID_SLAVE1 (FRAMEID_SLAVE0 + 2) //SecMem_Init() must be adapted
33 #define FRAMEID_SLAVE2 (FRAMEID_SLAVE1 + 2)
34
35 #define FRAMEID_EXTGT 7
36 #define FRAMEID_MBVEC 9
37
38
39 /* for testig purpose */
40 //#define FAULTY_SLAVE
```

```
41
42 #ifdef FAULTY_SLAVE
43 #define CYCLE_CORRECTION_OFFSET 7
44 #else
45 #define CYCLE_CORRECTION_OFFSET 0
46 #endif
47
48
49 /* indicate the frequency of the message; FREQUENCY denotes a message should
50  * be sent every FREQUENCY'th round */
51 #define FREQUENCY 10
52
53 /* please just use an constant (in bits) here, because this value is used by an
54  * external script to generate proper RSA keys
55  * maximum value: 254*8 = 2032 */
56 #define RSABITSIZE 256
57
58 /* maximum value: 254 bytes (RSA) */
59 #define FRAMEMBVECLEN (RSABITSIZE/8)
60
61 /* please just use an constant (in bits) here, because this value is used by an
62  * external script to generate proper RSA keys. vaild values are 128, 192 and
63  * 256 */
64 #define AESBITSIZE 256
65
66 /* this value should not be changed, since the blocksize is always 16 bytes for
67  * AES */
68 #define FRAMESLAVELEN 16
69
70
71
72 /* Master -> init membership system */
73 void SecMem_Init_Master(void);
74
75 /* Slave -> init membership system for slaves */
76 void SecMem_Init_Slave(void);
77
78 /* Slave -> call getGlobalTime(), encrypt the result and send it to the master;
79  *       => should be triggered at the beginning of a communication cycle! */
80 void SecMem_SendTimeToMaster();
81
82 /* Master -> receive the encrypted information, decrypt it;
83  *        => should be used as a callback function of receiving a frame! */
84 void SecMem_ReceiveSlaveTime(uint8 *encrypted, uint8 length, sint8 slaveno);
```

```
85
86 /* Master -> out of all received slave timing information, construct the
87  * membership vector, sign it and send it to the slaves
88  *         => should be triggered as soon as all slave timing information are
89  *         received! */
90 void SecMem_SendMembershipVector();
91
92 /* Slave -> receive extended global time information
93  *       => should be used as a callback function of receiving a frame! */
94 void SecMem_ReceiveExtGT(uint8 *data, uint8 length);
95
96 /* Slave -> receive the membership vector; if slave detects that it is out of
97  * membership, go in a safe state;
98  *       => should be used as a callback function of receiving a frame! */
99 void SecMem_ReceiveMembershipVector(uint8 *encrypt, uint8 length);
100
101 /* Master/Slave -> display membership vector (e.g. LEDs or UART) */
102 void SecMem_ShowMembershipVector(uint8 memvector);
103
104 /* encryption helper */
105 void init_rsa(void);
106
107 /* helper for increasing the ext. global time */
108 void increase_extended_global_time(void);
109
110 #endif
```

A.2. dispatcher.h

```
1 /*
2 Copyright (C) 2010   Bernhard Urban <lewurm@gmail.com>
3 Copyright (C) 2010   Sebastian Falbesoner <sebastian.falbesoner@gmail.com>
4 All rights reserved.
5 */
6
7 #ifndef __DISPATCHER_H
8 #define __DISPATCHER_H
9
10 #include "general.h"
11 #include "..\defs\_fr.h"
12 #include "Platform_Types.h"
13 #include "Flxr.h"
14 #include "uart.h"
```

```
15
16 /* start at first task although not all tasks are proccessed when a new cycle
17  * begins */
18 #define FORCE_NEW_CYCLE
19
20 /* ensure that all tasks are proceed, regardless if a new cycle begins */
21 //#define HANDLE_ALL_TASKS
22
23 #if defined(FORCE_NEW_CYCLE) && defined(HANDLE_ALL_TASKS)
24 #error "both configuration options can't be choosen at the same time"
25 #endif
26
27 /* basic prescaler for dispatcher */
28 #define DISPATCHER_PRESCALER 10
29
30 #define DISPATCHER_MAX_TASKS 8
31
32
33 /* initialize the dispatcher, which register a callback for
34  * the begin-of-cycle interrupt */
35 void dispatcher_init(void);
36
37 /* add a new task to dispatcher */
38 void dispatcher_addTask(uint8 idx, uint16 freq, void (*handler)(void));
39
40 /* remove an existing task */
41 void dispatcher_removeTask(uint8 idx);
42
43 /* this function is called periodically from the while(1)-loop in the main()
44  * function */
45 void dispatcher_loop(void);
46
47 /* register a callback function for increasing the extended global time */
48 void dispatcher_register_incgt(void (*f)(void));
49
50 #endif
```

A.3. perf.h

```
1 /*
2 Copyright (C) 2010   Bernhard Urban <lewurm@gmail.com>
3 Copyright (C) 2010   Sebastian Falbesoner <sebastian.falbesoner@gmail.com>
4 All rights reserved.
```

```
 5 */
 6
 7 #ifndef __PERF_H
 8 #define __PERF_H
 9
10 #include "Platform_Types.h"
11 #include "uart.h"
12 #include "reloadtimer.h"
13 #include <limits.h>
14
15 /* how many rounds should be measured for CPU load? */
16 #define PERF_ROUNDS 700
17
18 /* TICKS_PER_ROUND_CLEAN are the ticks measured without any application
19  * except perf.c running on the microcontroller (with PERF_ROUNDS=1000) */
20 #define TICKS_PER_ROUND_CLEAN (13331600ULL/1000ULL)
21
22
23 /* initialize perf -- reset values and add callback to dispatcher */
24 void perf_init(void);
25
26 /* function is called periodically by timer interrupt and calculates CPU usage */
27 void perf_cpucb(void);
28
29 /* function count up cycles; place a call the main loop */
30 void perf_countup(void);
31
32 /* show CPU usage on UART */
33 void perf_showCpu(void);
34
35 /* show memory usage (rely on our sbrk-hack) on UART */
36 void perf_showMem(void);
37
38 #endif
```

B. Source Files

B.1. membership_master.c

```
1  /*
2  Copyright (C) 2010   Bernhard Urban <lewurm@gmail.com>
3  Copyright (C) 2010   Sebastian Falbesoner <sebastian.falbesoner@gmail.com>
4  All rights reserved.
5  */
6
7  #include "include\membership.h"
8
9  static void init_aes();
10
11 static uint8 membership_vector = 0;
12 static psAesKey_t _aeskeys[NO_SLAVES];
13
14 /* provided by membership_common.c */
15 extern psRsaKey_t _rsakey;
16 extern volatile uint32 clock_ext;
17
18
19 /* necassary hack, because C doesn't provide higher-order functions... */
20 static void SecMem_ReceiveSlaveTime0(uint8 *encrypt, uint8 length) {
21     SecMem_ReceiveSlaveTime(encrypt, length, 0);
22 }
23 static void SecMem_ReceiveSlaveTime1(uint8 *encrypt, uint8 length) {
24     SecMem_ReceiveSlaveTime(encrypt, length, 1);
25 }
26 static void SecMem_ReceiveSlaveTime2(uint8 *encrypt, uint8 length) {
27     SecMem_ReceiveSlaveTime(encrypt, length, 2);
28 }
29 static void SecMem_ReceiveSlaveTime3(uint8 *encrypt, uint8 length) {
30     SecMem_ReceiveSlaveTime(encrypt, length, 3);
31 }
32
33 static void (*secmemcb[MAX_SLAVES])(uint8 *buf, uint8 buflen) =
34     {SecMem_ReceiveSlaveTime0, SecMem_ReceiveSlaveTime1,
35         SecMem_ReceiveSlaveTime2, SecMem_ReceiveSlaveTime3};
36
37 void SecMem_Init_Master(void)
38 {
39     uint8 i=0;
40     for(; i < NO_SLAVES*2; i+=2)
```

```
41          Flxr_RegisterIntHandler(secmemcb[i/2], FRAMEID_SLAVE0+i);
42      dispatcher_addTask(0, FREQUENCY, SecMem_SendMembershipVector);
43      dispatcher_register_incgt(increase_extended_global_time);
44
45      clock_ext = rand();
46
47      init_aes();
48      init_rsa();
49  }
50
51  void SecMem_ReceiveSlaveTime(uint8 *encrypted, uint8 length, sint8 slaveno)
52  {
53      if (length > 0) {
54          uint8 plain[FRAMESLAVELEN];
55          uint8 cycle, i;
56          uint16 macrotick;
57          uint32 recv_clock_ext = 0;
58
59          memset(plain, 0, FRAMESLAVELEN);
60          psAesDecryptBlock(encrypted, plain, &_aeskeys[slaveno]);
61
62          membership_vector |= 1 << slaveno;
63          for(i=0; i < 4; i++) {
64              recv_clock_ext |= plain[FRAMESLAVELEN-2-i] << (i*8);
65          }
66
67          Flxr_GetGlobalTime(ERAY_CTRLIDX, &cycle, &macrotick);
68
69          /* Is the slave evil and has manipulated global time? */
70          if ((cycle != plain[FRAMESLAVELEN-1]) ||
71                  (clock_ext != recv_clock_ext)) {
72              membership_vector &= ~(1 << slaveno);
73              dprintf("[secmem] cycle-counter mismatch, difference: %d\n",
74                      ((sint16)cycle)-((sint16)plain[FRAMESLAVELEN-1]));
75              dprintf("[secmem] extended-counter mismatch, "
76                      "difference: %d\n",
77                      ((sint32)clock_ext)-((sint32)recv_clock_ext));
78              dprintf("[secmem] clock_ext: %d\n", clock_ext);
79              dprintf("[secmem] recv_clock_ext: %d\n", recv_clock_ext);
80          }
81      }
82  }
83
84  void SecMem_SendMembershipVector()
```

```
85 {
86     static uint8 plain[FRAMEMBVECLEN];
87     static uint8 encrypt[FRAMEMBVECLEN];
88     uint32 elen = FRAMEMBVECLEN;
89     uint8 i;
90
91     /* send extended global time information in slot 7 */
92     memset(plain, 0, FRAMEMBVECLEN);
93     for(i=0; i < 4; i++) {
94         plain[FRAMEMBVECLEN-1-i] = (clock_ext >> (i*8)) & 0xff;
95     }
96     Flxr_TransmitTxLPdu(ERAY_CTRLIDX, FRAMEID_EXTGT, plain, FRAMEMBVECLEN);
97
98
99     /* send encrypted membership vector in slot 9 */
100    /* padding */
101    for(i=0; i < FRAMEMBVECLEN; i++) {
102        uint32 pad = rand();
103        memset(plain+i, pad, 1);
104    }
105
106    /* prevent "sanity/limit test"-fail; this actually happens if
107     * the data, interpreted as a number, is bigger than N */
108    plain[0] = 0;
109    plain[FRAMEMBVECLEN-1] = membership_vector;
110
111    psRsaCrypt(NULL, plain, FRAMEMBVECLEN, encrypt, &elen, &_rsakey,
112            PRIVKEY_TYPE);
113    Flxr_TransmitTxLPdu(ERAY_CTRLIDX, FRAMEID_MBVEC, encrypt, FRAMEMBVECLEN);
114    SecMem_ShowMembershipVector(membership_vector);
115 }
116
117 static void init_aes()
118 {
119     int32 aesret;
120
121     /* not that nice, but it get computed at compile-time which saves
122      * cpu-time and, a way more important, RAM */
123 #if NO_SLAVES >= 1
124     if((aesret=psAesInitKey(AES_KEY0,AES_KEYLEN0,&_aeskeys[0])) !=0) {
125         dprintf("AES: something was wrong with the key(0) initialization:"
126                 "%i\n", aesret);
127     }
128 #endif
```

```
129 #if NO_SLAVES >= 2
130     if((aesret=psAesInitKey(AES_KEY1,AES_KEYLEN1,&_aeskeys[1])) !=0) {
131         dprintf("AES: something was wrong with the key(1) initialization:"
132                 "%i\n", aesret);
133     }
134 #endif
135 #if NO_SLAVES >= 3
136     if((aesret=psAesInitKey(AES_KEY2,AES_KEYLEN2,&_aeskeys[2])) !=0) {
137         dprintf("AES: something was wrong with the key(2) initialization:"
138                 "%i\n", aesret);
139     }
140 #endif
141 #if NO_SLAVES >= 4
142     if((aesret=psAesInitKey(AES_KEY3,AES_KEYLEN3,&_aeskeys[3])) !=0) {
143         dprintf("AES: something was wrong with the key(3) initialization:"
144                 "%i\n", aesret);
145     }
146 #endif
147 }
```

B.2. membership_slave.c

```
1  /*
2  Copyright (C) 2010   Bernhard Urban <lewurm@gmail.com>
3  Copyright (C) 2010   Sebastian Falbesoner <sebastian.falbesoner@gmail.com>
4  All rights reserved.
5  */
6
7  #include "include\membership.h"
8
9  static void init_aes_slave();
10
11 static psAesKey_t _aeskey;
12
13 /* provided by membership_common.c */
14 extern psRsaKey_t _rsakey;
15 extern volatile uint32 clock_ext;
16
17
18 void SecMem_Init_Slave()
19 {
20     /* Slave Nodes perform getGlobalTime() at the begin of each communication
21      * cycle, encrypt the value and prepare sending in a slot of the remaining
```

```
22        * cycle */
23      dispatcher_addTask(0, FREQUENCY, SecMem_SendTimeToMaster);
24      dispatcher_register_incgt(increase_extended_global_time);
25
26      /* register callback for membership-vector */
27      Flxr_RegisterIntHandler(SecMem_ReceiveExtGT, FRAMEID_EXTGT);
28      Flxr_RegisterIntHandler(SecMem_ReceiveMembershipVector, FRAMEID_MBVEC);
29
30      init_aes_slave();
31      init_rsa();
32  }
33
34  void SecMem_SendTimeToMaster()
35  {
36  #define FID (FRAMEID_SLAVE0 + (SLAVE_NUMBER * 2))
37      uint8 cycle, i;
38      uint16 macrotick;
39
40      uint8 message[FRAMESLAVELEN];
41      uint8 encrypted[FRAMESLAVELEN];
42
43      /* padding */
44      for(i=0; i < FRAMESLAVELEN; i+=4) {
45          uint32 pad = rand();
46          memcpy(message+i, &pad, 4);
47      }
48
49      Flxr_GetGlobalTime(ERAY_CTRLIDX, &cycle, &macrotick);
50
51      message[FRAMESLAVELEN-1] = (cycle + CYCLE_CORRECTION_OFFSET) % 64;
52      for(i=0; i < 4; i++) {
53          message[FRAMESLAVELEN-2-i] = (clock_ext >> (i*8)) & 0xff;
54      }
55
56      psAesEncryptBlock(message, encrypted, &_aeskey);
57      Flxr_TransmitTxLPdu(ERAY_CTRLIDX, FID, encrypted, FRAMESLAVELEN);
58  }
59
60  void SecMem_ReceiveExtGT(uint8 *data, uint8 length)
61  {
62      uint8 i;
63      clock_ext = 0;
64      for(i=0; i < 4; i++) {
65          clock_ext |= data[FRAMEMBVECLEN-1-i] << (i*8);
```

```
66      }
67  }
68  void SecMem_ReceiveMembershipVector(uint8 *encrypt, uint8 length)
69  {
70      static uint8 plain[FRAMEMBVECLEN];
71      uint32 elen = FRAMEMBVECLEN;
72      int32 rsaret;
73      rsaret = psRsaCrypt(NULL, encrypt, FRAMEMBVECLEN, plain, &elen,
74              &_rsakey, PUBKEY_TYPE);
75
76      if((SLAVE_NUMBER < length &&
77              !((plain[FRAMEMBVECLEN-1] >> SLAVE_NUMBER) & 1))
78              || (rsaret == -1)) {
79          ; /* here we go into a safe state in a realworld application */;
80      }
81      SecMem_ShowMembershipVector(plain[FRAMEMBVECLEN-1]);
82  }
83
84  static void init_aes_slave(void)
85  {
86      int32 aesret;
87
88      /* not that nice, but it get computed at compile-time which saves
89       * cpu-time and, a way more important, RAM */
90  #if SLAVE_NUMBER == 0
91      const unsigned char *key = AES_KEY0;
92      uint32 len = AES_KEYLEN0;
93  #elif SLAVE_NUMBER == 1
94      const unsigned char *key = AES_KEY1;
95      uint32 len = AES_KEYLEN1;
96  #elif SLAVE_NUMBER == 2
97      const unsigned char *key = AES_KEY2;
98      uint32 len = AES_KEYLEN2;
99  #elif SLAVE_NUMBER == 3
100     const unsigned char *key = AES_KEY3;
101     uint32 len = AES_KEYLEN3;
102 #endif
103
104     if((aesret = psAesInitKey(key, len, &_aeskey)) != 0) {
105         dprintf("AES: something was wrong with the key(%i) initialization:"
106                 "%i\n", SLAVE_NUMBER, aesret);
107     }
108 }
```

B.3. membership_common.c

```
1  /*
2  Copyright (C) 2010   Bernhard Urban <lewurm@gmail.com>
3  Copyright (C) 2010   Sebastian Falbesoner <sebastian.falbesoner@gmail.com>
4  All rights reserved.
5  */
6
7  #include "include\membership.h"
8
9  psRsaKey_t _rsakey;
10 volatile uint32 clock_ext = 0;
11
12 void increase_extended_global_time(void)
13 {
14     /* a overflow isn't actually a problem.
15      * consider a cyclelenght of 8.8ms, therefore the first overflow happens
16      * after: 8.8ms * 64cycles * 2^32 = 76,7 years */
17     clock_ext++;
18 }
19
20 void SecMem_ShowMembershipVector(uint8 memvector)
21 {
22     static uint16 i = 0;
23     uint8 t = 0;
24     i++;
25     if(i == 5) {
26         dprintf("memvec(@node%d): %i|%i|%i|%i\n", NODE,
27                 (memvector >> 0) & 1, (memvector >> 1) & 1,
28                 (memvector >> 2) & 1, (memvector >> 3) & 1);
29         i = 0;
30     }
31     t |= ((memvector >> 0) & 1) << 4;
32     t |= ((memvector >> 1) & 1) << 5;
33     t |= ((memvector >> 2) & 1) << 6;
34     t |= ((memvector >> 3) & 1) << 7;
35
36     t = ~t;
37     IO_PDR14.byte = (t & 0xf0) | (IO_PDR14.byte & 0x0f);
38 }
39
40 void init_rsa(void)
41 {
42     int32 rsaret = 0;
```

```
43
44      rsaret |= pstm_init_for_read_unsigned_bin(NULL, &(_rsakey.N), 1);
45      rsaret |= pstm_init_for_read_unsigned_bin(NULL, &(_rsakey.e), 1);
46      rsaret |= pstm_init_for_read_unsigned_bin(NULL, &(_rsakey.d), 1);
47
48      rsaret |= pstm_read_unsigned_bin(&(_rsakey.N), RSA_N, ARLEN(RSA_N)-1);
49  #ifndef SLAVE_NUMBER
50      /* the master just needs the private key to sign the membership vector */
51      rsaret |= pstm_read_unsigned_bin(&(_rsakey.d), RSA_D, ARLEN(RSA_D)-1);
52  #else
53      /* just initialize the public key for a slave */
54      rsaret |= pstm_read_unsigned_bin(&(_rsakey.e), RSA_E, ARLEN(RSA_E)-1);
55  #endif
56
57      _rsakey.size = ARLEN(RSA_N)-1;
58      _rsakey.optimized = 0;
59
60      if(rsaret != 0) {
61          dprintf("RSA: something was wrong with the key initialization:"
62                  "%i\n", rsaret);
63      }
64  }
```

B.4. dispatcher.c

```
1  /*
2  Copyright (C) 2010   Bernhard Urban <lewurm@gmail.com>
3  Copyright (C) 2010   Sebastian Falbesoner <sebastian.falbesoner@gmail.com>
4  All rights reserved.
5  */
6
7  #include "include\dispatcher.h"
8
9  /* which task is active? */
10 static volatile uint8 _counter = 0;
11
12 /* task callbacks */
13 static void (*_tasks[DISPATCHER_MAX_TASKS])(void) = {NULL_PTR};
14
15 /* global variables to manage frequency of tasks */
16 static uint16 freq_limit[DISPATCHER_MAX_TASKS] = {0};
17 static uint16 freq_counter[DISPATCHER_MAX_TASKS] = {0};
18
```

```
19 /* interrupt flag */
20 static volatile uint8 _flag = 0;
21
22 /* callbacks function */
23 static void dispatcher_newcycle(void);
24 static void (*_increment_gt)(void) = NULL_PTR;
25
26
27 void dispatcher_init(void)
28 {
29     /* register a callback function on for the
30      * begin-of-cycle interrupt of the FlexRay Stack */
31     Flxr_RegisterBOCHandler(dispatcher_newcycle);
32 }
33
34 void dispatcher_addTask(uint8 idx, uint16 freq, void (*f)(void)) {
35     if(idx < DISPATCHER_MAX_TASKS) {
36         _tasks[idx] = f;
37         freq_counter[idx] = 0;
38         freq_limit[idx] = freq ? freq : 1;
39     }
40 }
41
42 void dispatcher_removeTask(uint8 idx)
43 {
44     if(idx < DISPATCHER_MAX_TASKS) {
45         _tasks[idx] = NULL_PTR;
46         freq_counter[idx] = 0;
47         freq_limit[idx] = 0;
48     }
49 }
50
51 void dispatcher_register_incgt(void (*f)(void))
52 {
53     _increment_gt = f;
54 }
55
56 static uint8 cycle_difference(uint8 newc, uint8 oldc)
57 {
58     /* valid cycle values are 0 to 63 */
59     if(newc >= oldc) {
60         return newc - oldc;
61     } else {
62         /* increment the extended global time */
```

```
63          if(_increment_gt != NULL_PTR) {
64              _increment_gt();
65          }
66          return (64-oldc) + newc;
67      /* examples:
68       * - oldc=63, newc=1 => return 2;
69       * - oldc=63, newc=0 => return 1;
70       */
71      }
72  }
73
74  static void dispatcher_newcycle(void) {
75      static uint16 prescale = 0;
76      static uint8 firstexec = 1;
77
78      uint8 new_cycle;
79      static uint8 old_cycle;
80      uint16 macrotick;
81      Flxr_GetGlobalTime(ERAY_CTRLIDX, &new_cycle, &macrotick);
82
83      /* begin-of-cycle interrupts can get lost, when the interrupt isn't handeld
84       * fast enough (which can happen quite often, depending on the FlexRay
85       * configuration) */
86      if(firstexec == 0) {
87          prescale += cycle_difference(new_cycle, old_cycle);
88      }
89
90      if(prescale >= DISPATCHER_PRESCALER) {
91          if(_flag == 1) {
92  #if 0
93              dprintf("[dispatcher]: please try a lower frequency\n"
94                      "              or modify DISPATCHER_PRESCALER\n");
95          }
96  #else
97          /* alternatively just use the 6th LED for output (in order to
98           * save some cpu time) */
99              IO_PDR14.byte &= ~(1<<5);
100         } else {
101             IO_PDR14.byte |= (1<<5);
102         }
103 #endif
104         _flag = 1;
105 #ifdef FORCE_NEW_CYCLE
106         _counter = 0;
```

```
107 #endif
108         prescale -= DISPATCHER_PRESCALER;
109     } else if(prescale < DISPATCHER_PRESCALER) {
110         /* why? to ensure all nodes start (nearly) at the same time */
111         if(firstexec) {
112             if(old_cycle < 60 && 60 <= new_cycle) {
113                 firstexec = 0;
114             }
115         }
116     }
117 
118     old_cycle = new_cycle;
119 }
120 
121 void dispatcher_loop()
122 {
123     if(_flag == 1) {
124         if(_counter < DISPATCHER_MAX_TASKS) {
125             if(_tasks[_counter] != NULL_PTR) {
126                 if(freq_counter[_counter] >= freq_limit[_counter]) {
127                     _tasks[_counter]();
128                     freq_counter[_counter] = 0;
129                 } else if(freq_counter[_counter] < freq_limit[_counter]){
130                     freq_counter[_counter]++;
131                 }
132             }
133 
134             _counter++;
135         } else if(_counter == DISPATCHER_MAX_TASKS) {
136             _flag = 0;
137 #ifdef HANDLE_ALL_TASKS
138             _counter = 0;
139 #endif
140         }
141     }
142 }
```

B.5. perf.c

```
1 /*
2 Copyright (C) 2010  Bernhard Urban <lewurm@gmail.com>
3 Copyright (C) 2010  Sebastian Falbesoner <sebastian.falbesoner@gmail.com>
4 All rights reserved.
```

```
 5 */
 6
 7 #include "include\perf.h"
 8
 9 /* functions from our sbrk-implementation */
10 unsigned short getMemUsage(void);
11 unsigned short getMemUsageMax(void);
12 long getBrksize(void);
13 long getBrkMax(void);
14
15 void perf_showMem(void)
16 {
17     dprintf("[mem%, mem%max; brk_siz, brk_max]:   %3d%%o, %3d%%o; %4d,"
18             "%4d\n", getMemUsage(), getMemUsageMax(), getBrksize(),
19             getBrkMax());
20 }
21
22
23 static unsigned long long cputicks, cputicks_max;
24
25 void perf_init(void)
26 {
27     cputicks = 0;
28     cputicks_max = ULONG_LONG_MAX;
29
30     /* reloadtimer4 is set to 100Hz */
31     init_rt4(perf_cpucb);
32 }
33
34 void inline perf_countup(void)
35 {
36     cputicks++;
37 }
38
39 void perf_cpucb(void)
40 {
41     static uint16 round = 0;
42     if(round == PERF_ROUNDS) {
43         /* lesser ticks ~ more load */
44         if(cputicks < cputicks_max) {
45             cputicks_max = cputicks;
46         }
47         /* display cpu- and memstats */
48         perf_showCpu();
```

```
49          perf_showMem();
50
51          /* reset values */
52          cputicks = 0;
53          round = 0;
54      } else if (round < PERF_ROUNDS) {
55          round++;
56      }
57  }
58
59  void perf_showCpu(void)
60  {
61      uint16 now = 1000 - ((uint16)((cputicks*1000ULL)/
62                  (TICKS_PER_ROUND_CLEAN * PERF_ROUNDS)));
63      uint16 max = 1000 - ((uint16)((cputicks_max*1000ULL)/
64                  (TICKS_PER_ROUND_CLEAN * PERF_ROUNDS)));
65      dprintf("[load%, loadmax%; ticks, ticks_max]: %3d%%o, %3d%%o; %10llu,"
66              "%10llu\n", now, max, cputicks, cputicks_max);
67  }
```