Nr.: FIN-008-2010

Reducing time and effort by concurrent firmware update processes on micro-controllers

Michael Schulze und Jörg Diederich

Arbeitsgruppe Eingebettete Systeme und Betriebssysteme

Nr.: FIN-008-2010

Reducing time and effort by concurrent firmware update processes on micro-controllers

Michael Schulze und Jörg Diederich

Arbeitsgruppe Eingebettete Systeme und Betriebssysteme

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Michael Schulze
Postfach 4120
39016 Magdeburg
E-Mail: mschulze@cs.ivs.ovgu.de

http://www.cs.uni-magdeburg.de/Technical_reports.html
Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 24.11.2010

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Reducing time and effort by concurrent firmware update processes on micro-controllers

Michael Schulze and Jörg Diederich

Institute for Distributed Systems
University of Magdeburg, Germany
mschulze@ivs.cs.ovgu.de

Abstract. Maintenance is a part of the software development process. In distributed systems, update actions require special attention in order to limit the effort. Several developments exist for sensor networks already. However, for less dynamic networks of micro-controllers the constraints are more simple. We exploit this to reduce the time necessary for an update and to reduce the resource consumption, which is essential in embedded systems. This paper presents our approach to update several micro-controllers simultaneously using the existing CAN communication properties. An evaluation of the developed prototype shows the benefits of our approach.

1 Introduction

Working with software for embedded devices is often a laborious job. During the whole software life cycle, a developer is confronted with hard constraints, limited resources and debugging possibilities. High efforts in time, material, money and so on are direct consequences.

These characteristics affect the replacement of a running program. Similar to other areas of software application, software developed for embedded systems often is under constant change. Functionality has to be corrected, added or removed due to new requirements, or simply because of fixing errors. With the availibility of rewritable program memory, update processes may take place during the whole life cycle of an embedded system. Every update may be even more simplified by using In-System-Programming (ISP). To move new program code via existing communication interfaces is a concept as well as a common feature of ISP. No additional interfaces have to be established for update purposes only. Accordingly, a removal of an embedded system from its environment can be avoided.

Today, ISP is established in single systems. Using embedded systems in a distributed system challenges the update procedure again. Most existing ISP approaches require to establish and release an individual connection to each participant. More advanced approaches connect to a distributed system itself. Afterwards, all steps of ISP are applied to each participant individually. In our opinion, it is insufficient to adhere at existing procedures or only apply them. We claim that even for a small numbers of participants, the steps to update multiple embedded systems at a time have to be more customized and comfortable. By using broadcast capabilities of a communication infrastructure, multiple participants may be updated in parallel. It would be a waste of time to deal with each embedded system individually. In case a physical connection needs to be established and released individually, stress of material and humans make matters even worse.

A limitation is that all contacted participants update simultaneously to the same software. However, it can be expected that several embedded systems working together are intended to run identical software. To reuse development results whenever possible reduces effort and eases maintenance. In large sensor networks it is almost impossible to provide each sensor with individual software. However, for small numbers of affected systems, running identical code is sufficient. In order to illustrate this, consider our example of a mobile robot system (see Fig. 1). It consists of several motors and sensors, controlled by some micro-controllers forming a distributed system. Acceleration and driving algorithms are identical for all wheels. Different outputs at each wheel of the robot only occur if the inputs differ. The same applies to sensor control algorithms. While being attached to different micro-controllers, equivalent sensors, like for instance distance sensors, will preferable being controled by equal algorithms. Assuming all controllers communicate with each other via a field bus like CAN (Controller Area Network) [14], an update could be run in parallel. Like software development and enhancement, all reprogramming steps have to be done only once.

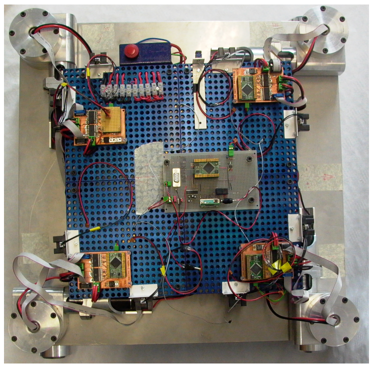

Fig. 1. The Q transport platform – at each corner a micro-controller controls a driving and a steering motor.

We believe all features a network of embedded system provides have to be used in order to support the update process as much as possible. Broadcast capabilities of already existing communication infrastructures enables us to connect to multiple embedded systems at a time and work in parallel. We present our approach of using the CAN field bus communication for an update process of embedded systems. Instead of the sequential order in other approaches, we make full usage of the CAN possibilities to perform a simultaneous distribution of new program code.

The rest of the paper is structured as follows. Existing procedures and technologies in firmware updates are described in section 2. Our approach is explained in detail in section 3. The results of a performed evaluation are content of section 4, together with a description of the used environment. Section 5 summarises our work and describes reasonable improvements and therefore gives an outlook to future research.

2 Related work

The usage of micro-controllers gives embedded systems a high degree of flexibility. The same kind of controller may be used for different purposes, simply by running different software on it. Sometimes, it controls a number of sensors and sometimes it is responsible of relaying messages. Once in operation, the ISP allows to perform maintenance functions easily.

The update or programming process is a cooperation of at least two parties. One participant, commonly called the host, distributes the changes. Micro-controllers who use the changes to modify their installed software could be called clients. The host is typically presented by a commonly used personal computer. As software development for embedded systems is done on such systems, it is obviously to use them for update purposes too.

Existing aproaches of the ISP rely on a simple relation between the host and a client. At any time, the programming software used at the host communicates with exactly one client. A communication with another client requires releasing an existing connection first, either physically or logically. Figure 2 illustrates this relationship with one host and four clients. Present developments differ in the used algorithms on the client sides and the applied communication connection only.

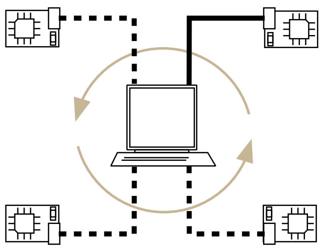

Fig. 2. One-to-one relation between a host and four clients

In order to ensure an always exisiting possibility to update, manufacturers of micro-controllers often provide hard-wired algorithms. For each architecture, the details depend on the manufacturer. The algorithms widely included in the AVR micro-controllers of Atmel [4] give an example. Connected via the Serial Peripheral Interface (SPI), the host commands the micro-controller in the update process. A selection of a dedicated client is not intended. The connection between host and client is a straight cable, and any intention to select another client results in a release and new attachement of the wires. This is similar to another example of hard-wired algorithms, the monitor mode of HC08 micro-controllers made by Freescale [10]. Besides its use for debugging processes, selected monitor commands allow to update the persistent flash memory. As in the AVR example, the communication is done using a serial interface. However, the kind of communication follows no standardization. A single pin is used in the connection between host and micro-controller. Again, a selection of a certain client is not intended, even with multiple clients connected to the wire. Due to the lack of an appropriate flow control, a communication with multiple participants is not possible at all.

Hardware solutions allow to program micro-controllers even in case of a blank flash. Software algorithms have to be present in flash instead. By using small programs that are able to write into persistent memory, restrictions given by hard-wired algorithms can be avoided. Out of the available communication interfaces, a selective support is possible. Individual protocols may be designed for the intended purpose, too. Additionally, the update algorithms themselves can be extended, corrected or globally changed during the life of a micro-controller. In order to resolve its task, an update software is the first application starting right after the reset. It decides about booting in terms of starting another application, or to prepare an update process. Therefore, programming software situated on micro-controllers fulfills the criteria of bootloaders.

The number of available bootloaders for micro-controllers is vast. Hardware manufacturers support developments for their products with own proposals. Again, Atmel gives an example with their implementation using a serial UART interface [2]. There are a number of open-source developments with this functionality, too [11, 15, 1]. As already explained, any bootloader using strict end-to-end connections like UART or SPI is not suitable for programming multiple recipients in parallel by design. Developments making use of bus interfaces promise to be more adequate instead. In [3] Atmel proposes an bootloader, both for CAN and UART interfaces. By using the connections already existing, clients attached to the CAN bus do not need to be additionally connected to the host anymore. However, the protocol used in communication does not support a simultaneous update. Request and response data use equal identifiers. The relation between message identifier and content, as recommended in the CAN standard, has not been paid attention. In direct consequence, the correct course of an update is likely to get lost. Clients may interpret answers originated by other clients as requests given by the host. Due to this fact, only a sequential processing of attached clients is provided.

For ethernet as the most popular bus standard, the Ethernut project [6] provides a bootloader. Performing in opposite directions, each booting client may request an update via the TFTP protocol. The decision for an update is given by the DHCP request made before. A simultaneous boot of micro-controllers may result in multiple update processes running in parallel. In direct comparision with CAN, the used ethernet interface is less common to small micro-controllers. It requires more resources and therefore more components need to be included on the board. Additionally, each update process results in a new transmission of data and is not parallel at all. Other, most likely equivalent transmissions will be ignored. Finally, the active behaviour of the micro-controllers does not allow any verification operations by default. Only in case an update was successful, the host is able to request a single client for its flash memory content.

Our vision is to take advantage of a simultaneous update process with a minimum effort of additional resources. A connection between host and clients via the CAN bus requires little additional equipment at each client. In case CAN is already used in communication, even no additional effort is necessary at all. The field bus feature enables to distribute an update to all recepients simultaneously, which leads to a reduction of time and effort, as desired. By adapting a programming software used in traditional approaches, exisiting work flow may be continued.

3 Our approach

Connecting host and micro-controllers through the CAN bus supports our intention of a simultaneous update. Typical for the CAN field bus, sent data reach either all nodes attached to the bus or none of them. Each communication is a broadcast by definition. There is no need to perform any additional or special steps to communicate with all nodes. Figure 3 presents the structure of the connection between all participants. Like in common proceedings, the host triggers the update process. It is connected to all micro-controllers through the wires of the CAN bus. Each micro-controller is able to communicate with any other participant via the bus as well.

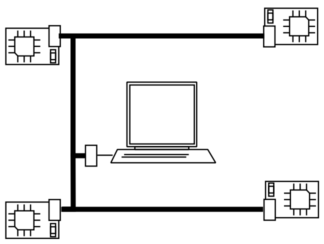

Fig. 3. Connection between host and micro-controllers via CAN

Besides the connection, the programming algorithms running at the host and all receiving nodes need to work together. The classical collaboration of programming software and bootloader appears to be well suited even in case of multiple active bootloader instances. Any requirements to identify single micro-controllers may be still left open, either for the programming software or the bootloaders. Having programming software, bootloader and communication messages, three logical elements exists in our update scheme. Each of them is illustrated in Figure 4. At any time, only one programming software is active. It is running at a dedicated node, the host. Communication to connected micro-controllers will be done by sending CAN protocol messages. Bootloaders running at the micro-controllers process received messages and react accordingly.

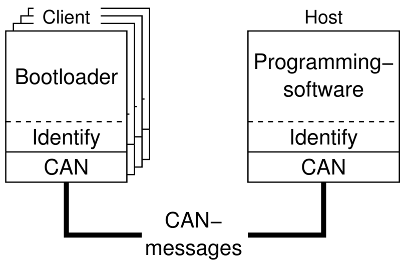

Fig. 4. Logical elements in the simultaneous update process

In theory, the simple task for the programming software is to transmit the new program code within CAN messages. At reception, each bootloader performs appropriate write operations to the persistant flash memory of its node. However, a feedback by the receivers is necessary in order to detect difficulties which may have occured. For instance, a requested write operation may have failed due to flash difficulties. The possibility of feedback requires a clear distinction between receiver feedback and host command messages. It can be achieved by a mapping between feedback messages and receivers. Knowing the choosen mechanism, one can easily associate a message to its originator. This enables fault node identification and, even more important, prevents a misinterpretation of feedback messages as host command messages at once. In section 2 we presented the difficulties with feedback messages of ATMELs proposed solution. Here the overall procedure was limited to perform only sequential updates.

For a mapping of messages, we assume that each node has an identifier – the 'node Id'. At runtime, a node id could be obtained by using election algorithms. With hard resource constraints even in update processes, a more simple proceeding is preferable. To either process hardware switches or to use data stored in other persistant memories is both efficient and sufficient.

The node Id is transmitted in the header of each CAN message. No other content is placed inside the header. Since they are independent systems, multiple nodes may start to transmit messages at any time. Without any constraint to all node Ids, if two or more nodes begin transmissions at the same time, equivalent headers could be sent in case of equivalent node Ids, leading to CAN specification violations. A uniqueness of each node Id will ensure conformance. During the assignment of each node Id, this additional constraint has to be met as well. Being a container for the node Id, the message header determines the sending node. A command associated with a message is also necessary, as for message interpretation and processing purposes. Leaving the header to the node Id, there are eight data bytes left to choose from. In our approach, each command will be described by an identifier in the first byte of the message content.

The described data placement - a node Id in the header, message command and data in the message payload - is advantageous in using existing filter mechanisms. A common CAN controller provides a hardware filter, which enables to ignore messages already by their header. This allows any receiver to reject irrelevant messages sent from other receivers. During the update dissemination, the host and all clients are message receivers. While knowing the host identifier, the micro-controller of clients will only be interrupted or waken up from power save modes by messages sent by the host. The latter can easily set up a filter for his expected clients, too.

In summary, the host filter is a collection of the inverted filters of all connected clients. Unlike client filters, which can be set up with a connect message immediatly, a host filter can not be set up that simple. Being connected to the CAN bus, not every message received during update initialization may be of interest. These messages of normal bus traffic have to be detected by comparing their content with the template of expected answers. The host filter mechanism requires two more facts to take care during initialization. First, in case command and reply content are equal, there is no possibility to distinguish commands sent from other hosts from expected replies of clients. From the host's point of view, such an initial message of another host is a reply of a client ready for an update. Second, responses of clients to initialization requests of other hosts should not be processed either. All possibilities described before will, if not taken into account, add node Ids to the filter which are invalid. Messages using these node Ids will not be dropped and thus disturb the update process, if disseminated. Therefore, we assume at most one host activating clients at any time. Requests and responses of other starting update processes will not occure and take influence. For the rare event of bus messages which accidently look like update messages, there is no such simple solution. The update proceeding has to handle the wrong filter setting by itself.

Looking at CAN, a message header should describe the message content it goes ahead. It should not act as an identification of the sender or the designated receiver. In the way described before, the data placement does not follow this intention. However, going the strict way and ignoring existing possibilities results into additional effort and caution. Only by selecting well-designed, disjoint command sets in requests and replies, receiving nodes are able to make usage of hardware filter possibilities. A filter can not be set up and reject host or client messages otherwise. In case multiple hosts are available, all command sets have to be different as well. Furthermore, if the message header is already reserved by a command, a sender identification has to be placed in the message content. As in CAN, at most only eight bytes are available here. In order to use most of these bytes for payload data, identifiers should without a doubt be as short as possible. Nevertheless, the performance of the data transport decrease. Additionally, the performance of message processing decreases, too. Each receiver has to process the message content to identify the sender. Under the constraints of limited resources, ignoring the given disadvantages can not be explained with a recommended data placement. Placing identification data in message headers is more advantageous. Message processing is simple, fast and straightforward. In protocol development, the placement offers more flexibility. Node identifiers reasonably may be expected to be stable, due to storage in e.g. EEPROM. In contrast, command identifiers will very likely change during development. Using them for filter purposes requires to check the whole command set during changes again.

The described message structure allows to perform the update procedure known from existing approaches with only small modifications. The following steps have to be taken:

1. Connection to nodes
2. Selection of memory to work on
3. Erase of memory content
4. Transmission of code update
5. Verification of updated memory content
6. Release of nodes

During step 1, the host activates all pending bootloaders. It sends a connect message using its identifier. All receivers will reply using their unique identifiers as well. More important, after processing a connect message, the respective bootloaders will only accept messages with an identifier of the activating host. During the following update process, any other messages having a different identifier will be ignored. This includes replies for the connecting message or other connecting messages, too. From now on, the programming software running at the host can safely command its associated nodes. As already mentioned before, each message sent contains a byte of command data. Originated from the host, a command may be followed by more data describing the command arguments. The selection of memory type during step 2, the address selection or the transmission of code during step 4 are examples of command arguments. In the other direction, each reply of a bootloader contains, in addition to the command byte, a return state of the process the command released. Consequently, it could be considered as a kind of RPC. However, there is one exception. During the verification process listed in step 5, each bootloader transmits selected memory contents to the host. This usually includes more data than one message is able to carry within. Hence, data fragmentation is necessary. Each node independently transmits the selected memory content with multiple messages, each using seven bytes of payload. Only the last byte of the last message identifies the return state of the executed read operation. Having one more byte for data in all previous messages reduces transmission overhead by one seventh. Of course, in case the memory content fills the last message exactly, one additional message has to be sent. For all verification transmissions, the unique message identfiers ensure the CAN protocol will not be violated. Finally, at step 6, the host finishes the update process by releasing the connected bootloaders. Each of them clears its identifier filter, therefore presenting a clean state to the following operations.

Working with multiple recepients requires the programming software to perform additional managing tasks. Instead of reacting to a simple reply, several and possibly different answers have to be respected. The identification of the originator with the message header, already described, is helpful again. On connection to micro-controllers, the programming software collects the identfiers sent with reply messages. A missing reply of one or more receivers can easily be detected during further steps. For this, every node replying is removed from a list of expected answers. In case the list is empty at last, the received answers need to be processed further. If the connected clients unanimously report success, the update process may proceed. In return, missing or diverging answers interrupt an update. All still connected nodes are released and the update process is finalized. To manage all clients, a solution is needed that handles invalid clients also. If the host message filter has changed unnecessarily due to bus messages that have been misinterpreted, the managing process expects more clients then exists. However, those ghost clients will not response appropriately to further host requests. Consequently, the update process will fail with first requests. The list of expected answers still contains Ids of nodes, which have not answered in time. With a host stopped at this step, one or more clients were probably already left activated. They will wait for a new update process advertised under the same node Id. If one host finally starts this process, they will act like new activated clients.

4 Evaluation and Results

Keeping the use case and the evaluation scenario close together allows to directly deduce terms of everyday usage. Figure 1 presents our Q-Robot use case, a transport platform controlled by four AT90CAN128 micro-controllers. Each of them is embedded in a single board. Connected via CAN, all micro-controllers are able to communicate with an attached personal computer. Of course they are able to communicate with each other, too.

Besides of missing actuators and sensors, our evaluation scenario is identical to the use case. With models from PEAK-System Technik [12], both use the same adapters necessary to connect a personal computer to the CAN bus. The choosen transmission rate of 250 kBit/s is also equivalent. From available adapters, the PCI adapter slot card has been selected. At the opposite end, Crumb128-CAN boards made by chip45 [5] represent the nodes. The transport platform is equipped with boards of the same model. For evaluation, a self-designed motherboard is additionally used. It integrates all nodes at once.

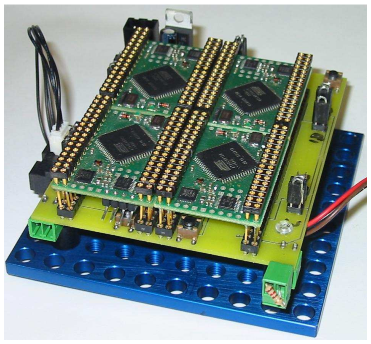

Fig. 5. Motherboard equipped with four Crumb128-CAN boards

Figure 5 shows our evaluation board. The plugged Crumb128-CAN boards are wired directly. Each of them may be reset or disconnected from power individually. Without having several wires and switches, evaluation is more easy and flexible.

On the side of software, the evalution environment corresponds to our use case. At the host, the server process is represented by a programming software, which communicates to the CAN bus via the PEAK CAN hardware adapter. All low-level accesses to the adapter are handled by software provided by the adapter manufacturer. Using the Linux operating system, the software consists of the *libpcan* library and the hardware driver. During evaluation, both were available with version 6.7 of the driver package [13]. As mentioned in section 3, the programming software has to connect to the CAN library, as well as to manage multiple clients. There is no existing software known which fulfils this requirements. Due to this, we extended 'Avrdude' [7], an already established programming software available in revision 5.3. In the terminology of Avrdude, a new programmer has been added by the implementation of our communication protocol. Thus, a simultaneous update is supported by Avrdude for the first time. Implementing an additional programmer is advantageous for evaluation purposes. Since other programmers are still available, a direct comparision of different approaches will be possible.

The communication partners of the host are bootloaders running at connected micro-controllers. Having C++ available for their implementation, our development could include object-oriented elements and low-level instructions at once. For the default Linux compiler GCC [9], a port called 'avr-gcc' exists. It provides cross-compiler possibilities to the AVR architecture for both C and C++. Closely related to the compiler, a pendant of the default C library exists, too. The *avr-libc* [8] contains standard routines and, in being designed for an embedded system, several special constructs, for example in order to service interrupts or to perform register accesses. As a result of design and implementation processes, a prototype of a bootloader became available. It supports the

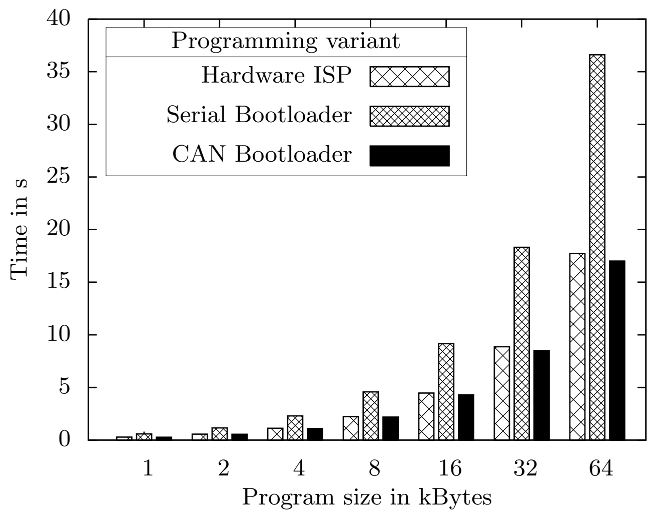

Fig. 6. Micro-controller programming times of different programming methods for an update on one client

requested update functionality for flash and eeprom memory. The version 4.1.1 of the avr-gcc cross-compiler and version 1.4.5 of the avr-libc library were used for the development of the prototype. The bootloader prototype requires eight kilobytes of flash memory at the bootloader section on the micro-controller.

This bootloader prototype is used in the evaluation process. For users, time necessary for an update is probably the value of most interest. Avrdude itself prints out time values during programming progress visualization. Although progress could be indicated individually by any programmer, known programmers act comparable. This allows using the final time value presented for speed estimations. Update time is measured for our approach with one, two and four clients programmed in parallel. For comparison purposes, measurements where done for two already existing approaches. The selected boards provide an interface to hard-wired programming algorithms (ISP) of the micro-controller. Having an adapter for the parallel port of the host, update time is also measured for this processing. In order to complete evaluation, time measurement is done for another software solution, a bootloader for serial communication [11]. The software is a traditional bootloader using a serial RS-232 connection in order to communicate with the host. Figure 6 illustrates the update times for different code sizes for the described variants and updating of one client only.

The values obtained give an impression of the benefit of our approach. In a one-to-one relation between host and client, our prototype requires less time than both concurrent approaches. The programming time with hardware algorithms is almost equal to our approach. Since flash circuit programming time is invariant,

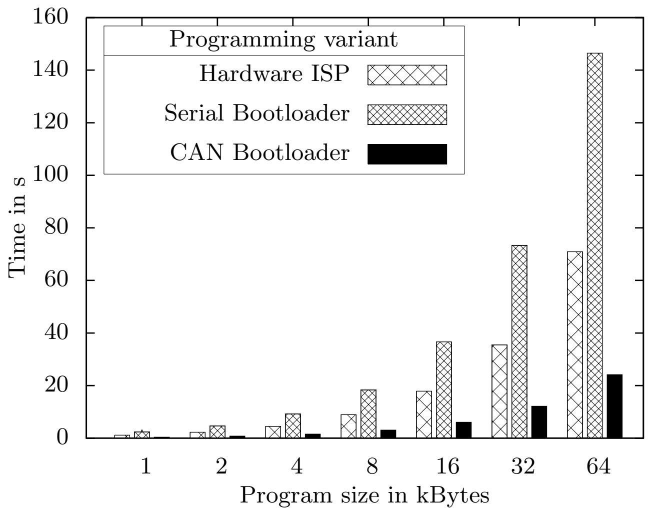

Fig. 7. Micro-controller programming times of different programming methods for an update on four clients

similar transmission and management qualities can be assumed. Time savings in comparision to the serial bootloader can be explained with a higher transmission rate of the used CAN bus.

Looking at the values for four clients shown in Figure 7, the advantage of our prototype becomes obvious.

The updating times for the hardware ISP and for the serial bootloader are calculated. We multiplied the needed time for one client update with four. That estimation is very optimistic at all. We do not take into account the needed time for changing the physical connection and restarting the updating tool again for each client. The really required times are probably a magnitude higher. Certainly, with our approach updating multiple clients requires additional effort too and it leads to higher update times in comparison to updating one client only. However, the increase in case of a parallel update is only a fraction of the time of one client and is very small in relation to the other methods.

The evaluation results illustrate the benefits of our prototype environment. Although developed for networks with multiple attached micro-controllers, no loss in comfort and usage exists even in an unicast scenario. Time savings increase linear with more and more micro-controllers becoming targets of the same program code.

5 Conclusions

Environments using multiple micro-controllers are different to the single and isolated approach. Supporting individual programs becomes more unhandy and more expensive with each additional controller. This can be avoided by an identical program running anywhere if possible, depending on the use case.

The need to update the program is faced with changed conditions, too. A traditional approach to update all micro-controllers program memory sequentally is inappropriate. Program memory access has to be created for the same program over and over again, which is difficult, if not impossible at all. This changes with network interfaces that became available. Attaching a micro-controller to a network establishes a persistent connection to updates. It also opens possibilities to use more advanced programing proceedings.

We presented our solution of a simultaneous update throughout a network of micro-controllers. Simplicity and retention of known workflows were main objectives in development. As usual, new program code is disseminated by a single node of the network. For update processing, we extended an existing and well proven programming software. The existing user interface was left unchanged at default level. Therefore, the existence of multiple update receivers is transparent to the user. Update processing is still possible for one attached micro-controller.

Update data transportation is done via CAN, a field bus which has its origins in the embedded world. The choice of this established and widely accepted bus prevents any needs of additional equipement from bigger worlds. Receiving micro-controllers are programmed in parallel due to broadcast properties of messages in CAN. As shown in the evaluation, our prototypical implementation is already comparable to the update time of single micorocontrollers. Considerably more important, update time increases only by a fraction with each additional receiver. The time savings can be confirmed by our experiences in everydays usage of the prototype. Also with the comfort of using an unchangeable programming process, the development presented met our expectations already at the first try.

Having a prototype running, future development can focus on improvements and optimizations. Perhaps the most significant problem existing is flash consumption. For the kind of micro-controllers intended, bootloader program size is limited to eight kilobytes. The current implementation takes almost this size. With only a few bytes reserved, the usage of the prototype will most likely depend on compiler optimization processes. Further development work is necessary to reduce the code size while preserving current functionality as well. The review should include speed issues, too. Besides of pure implementation improvements, update time could be reduced by using more payload in transfered messages. One precious byte of message data is currently used for specifying a message command. Describing the command within the message header increases payload by more than ten percent. Currently, a message header is used for identification purposes only. Most of the 29 bits of the included extended identifier are unused in case of commonly choosen low node identifiers. Therefore, reducing the allowed identification numbers by some bits provides room for the message command without a restriction in usage.

Adding more content to a message identifier requires an adaptation of filter algorithms. At the host, changes in all implemented CAN adapters of the programming software are necessary. As soon as different interfaces have to be used, a more general approach is advantageous. For CAN only, a first step could include a CAN controller abstraction inside of the programming software. Being more general often means to be more challenging, but provides independance of hardware and software interfaces, too. Our development FAMOUSO [16, 17] already separates application interfaces from interfaces of transport mechanisms. In further developments of the middleware, we intend to make this possibility available to update processes for the reasons described. We expect additional benefits by using the publish-subscribe paradigma FAMOUSO follows, which will enable different update publications with different receivers as well. In between, we look forward to include the presented solution in the regular Avrdude development branch.

References

1. Alexander Neumann. foodloader (Atmel Bootloader). Website: http://www.lochraster.org/foodloader/. revision 2008-05-11.
2. Atmel Corporation. *AVR109: Self Programming*, Rev. 1644G-AVR-06/04 edition.
3. Atmel Corporation. *AVR914: CAN & UART based Bootloader for AT90CAN32, AT90CAN64, & AT90CAN128*, 7592b-avr-01/06 edition.
4. Atmel Corporation. AVR 8-Bit RISC. Website: http://www.atmel.com/products/avr/default.asp, 2008. revision 2008-03-11.
5. chip45 GmbH & Co. KG. Mikrocontroller module "crumb128-can". Website: http://www.chip45.com, 2008. revision 2008-04-01.
6. egnite Software GmbH. Ethernet Boot Loader. Website: http://www.ethernut.de/en/eboot/index.html. revision 2008-03-11.
7. Free Software Foundation. Avr downloader/uploader. Website: http://www.nongnu.org/avrdude/, 2006. revision Fri Sep 23 23:21:06 MET DST 2005.
8. Free Software Foundation. Avr c runtime library. Website: http://www.nongnu.org/avr-libc/, 2008. revision 2008/04/20.
9. Free Software Foundation. Gcc, the gnu compiler collection. Website: http://gcc.gnu.org/, 2008. revision 2008-05-03.
10. Freescale Semiconductor, Inc. HC08. Website: http://www.freescale.com/webapp/sps/site/overview.jsp?nodeId=01624684497663, 2008. revision 2008-03-11.
11. Martin Thomas. AVRPROG compatible bootloader for ATMEL ATmega Controllers. Website: http://www.siwawi.arubi.uni-kl.de/avr_projects/index.html#avrprog_boot. revision 2008-05-10.
12. PEAK-System Technik GmbH. Pcan hardware. Website: http://www.peak-system.com, 2008. revision 2008-04-01.
13. PEAK-System Technik GmbH. PCAN Hardware – Linux Driver Page. http://www.peak-system.com/linux/index.htm, 2008. revision 2008-04-01.
14. Robert Bosch GmbH. *CAN Specification Version 2.0*, September 1991.

15. Roland Riegel. boofa: bootloader for Atmel AVR microcontrollers. Website: `http://www.roland-riegel.de/boofa/`. revision 2008-05-11.
16. Michael Schulze. FAMOUSO project website. `http://famouso.sourceforge.net`, 2008.
17. Michael Schulze. FAMOUSO – Eine adaptierbare Publish/ Subscribe Middleware für ressourcenbeschränkte Systeme. *Electronic Communications of the EASST (ISSN: 1863-2122)*, 17:12, 2009. Workshops der Wissenschaftlichen Konferenz Kommunikation in Verteilten Systemen 2009 (WowKiVS 2009).