Security aspects in CANopen bootloaders

Christian Keydel, Embedded Systems Academy

One intriguing aspect of networked nodes is the option to allow their firmware to be remotely updated in the field. Some CANopen Device Profiles have even made this a requirement. The updating process in the target is handled by a dedicated piece of firmware, the CANopen bootloader. A failure of the bootloader can have severe consequences, from necessary power cycles over direct mechanical interaction with the device up to having to replace the node. For deeply embedded nodes that are out of reach, for example in deep-sea applications, such a failure can even be catastrophic. Therefore, in this crucial piece of software, security and reliability of operation deserves special consideration. Combined with resource constraints common in bootloaders, these unique requirements ask for a dedicated rather than a common off-the-shelf CANopen implementation. This paper discusses the security aspects of such an implementation in regards to software engineering, safe coding practices and operation flow.

While every application is different in terms of what levels of reliability and security are deemed sufficient in a bootloader, this paper will attempt to discuss these aspects in a manner that should apply to a great number of fields. We'll start with some basic principles on the design of a bootloader.

The bootloader always executes first

No application should ever be able to interfere with the bootloading functionality since it is also a rescue mechanism. Therefore, after a reset of any kind, first the bootloader must gain control of the microcontroller to determine what it has to do. This means that the reset vector in the microcontroller must point to the entry address of the bootloader at all times. In devices where the reset and interrupt vectors are at fixed address location in flash memory, this imposes a slight challenge as commonly this region can only be erased and reprogrammed as a whole. Since erasing this area would create a window of insecurity during which a reset or power failure renders the device dead-in-the-water (sometimes literally), this is unacceptable. Instead, all vectors will have to be preset to fixed, determined locations.

Interrupt vector mirroring

The application code that the device will be executing will want to use interrupts, so we have to establish a method to use them, even if the actual vectors are fixed. This is possible by using mirroring, where the interrupt vectors point to fixed addresses in the loadable application

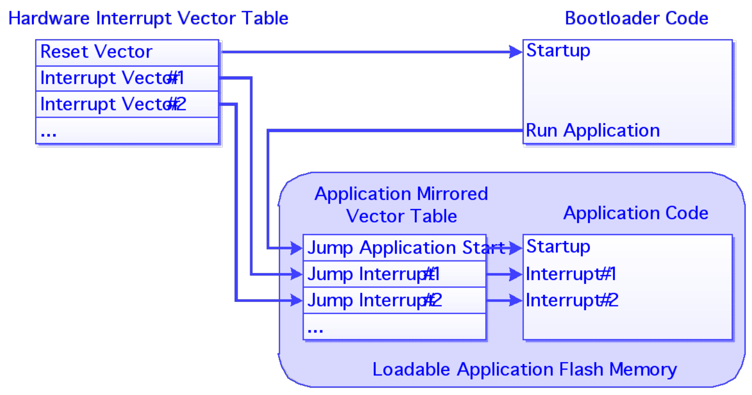

Figure 1 – Interrupt Vector Mirroring

area, which in turn contain jump instructions to the actual interrupt service routines (ISRs), as shown in Figure 1.

It should be noted that with microcontrollers where the vector address location is programmable, mirroring can be avoided. The bootloader can use the reset vector for itself and set the vector base address to the application's vectors before it starts executing it.

No use of interrupts

Unlike the application, it is preferable for a bootloader not to use interrupts for several reasons: When vector mirroring has to be employed, using the hardware vectors for the bootloader is impossible unless all vectors point to code that decides which ISR (the bootloader's or the application's) to execute, which introduces additional interrupt latency and also a potential point-of-failure during execution. Secondly, even when mirroring is not used, interrupt-triggering events occur asynchronously to the rest of the code and the execution of the code is no longer completely deterministic which for the highest level of security has to be avoided. Lastly, with most architectures the microcontroller cannot execute ISRs during flash programming which means interrupts have to be temporarily disabled during programming sequences. The mechanisms to accomplish this are not always absolutely reliable and may create another potential point-of-failure.[1]

Using polling

The performance requirements of a CANopen bootloader are typically very modest and can be achieved without the use of interrupts, using loops and polled events instead. There is very little potential gain in using interrupts in a bootloader outside of the ability to re-use standard drivers taken from regular applications.

Flash programming and security

The ability to re-program the flash memory area reserved for the application is the core functionality of a bootloader. It typically consists of three parts:

- Erase
- Program
- Verify

While the code to verify successful programming executes only read accesses and is therefore uncritical, erase and program alter the flash contents and can render a device useless if executed inadvertently. This may happen if an application or hardware bug, EMI or power disturbance causes the microcontroller to execute random addresses. There are three levels of security against this type of transient failure causing permanent damage:

First, the code to erase and program flash memory should not be scattered throughout the bootloader but instead be minimized in size and placed at as few locations as possible. In some cases, the code to erase and program looks almost identical save for some parameters. In this case, one should take advantage of this and implement both inside a single function.

In addition, if a higher level of security is required, the lowest-level flash programming code should be guarded by a flag pattern or "password" at a certain memory location that only the bootloader sets after it has started up successfully and far enough away from the execution path that forcibly ends up in executing the programming function. The password will be destroyed upon application execution. This helps against inadvertent execution of the programming code while the application is running.

For the highest level of security, the programming code should not exist in the flash memory at all. Instead, if the microcontroller architecture allows this, it should be downloaded into RAM only right before the application code is to be programmed and automatically destroyed afterwards. In a CiA 302 compatible bootloader, a dedicated entry in the objects $1F50_h$ (Program data), $1F51_h$ (Program control) and the other associated arrays ($1F56_h$ etc.) offers a

[1] See Philips Semiconductors AN10414 – Handling of spurious interrupts in the LPC2000, chapter 2, for an example

convenient interface to implement this.[2] It should be noted, however, that the version management part of a CANopen manager application dealing with such a slave node may need to be customized in order to support this scheme properly. As an alternative, the flash programming code could also be embedded in an application firmware file so that it is always downloaded together with the application code and automatically placed in RAM by the file parser in the bootloader before the actual programming starts.

Protection against unauthorized access

It can be very desirable or even necessary to offer the option to update nodes in the field to authorized parties only. With CANopen and CiA 302 being an open standard, this by default is not guaranteed. Therefore, a custom write-only object dictionary entry that unlocks the flash programming in the software only when the correct password is written to it, or a more sophisticated challenge/response mechanism to prevent eavesdropping the password may be implemented. This type of mechanism is

Bootloader overwrite protection

Before a received block of application code is programmed into flash memory, the bootloader must check for the block to stay completely within application flash area boundaries. Any attempt to program flash outside these boundaries must be rejected and an error must be generated, e.g. via an SDO abort message.

In addition, if the hardware supports it, the bootloader flash area should be protected against erase and programming by activating a flash locking mechanism. Thereby, even if for whatever reason an erase or programming cycle is triggered, the bootloader will always stay functional.

Checksum calculation and verification

Any kind of programmable data, be it the application code, configuration data in an EEPROM or the flash programming code that goes into RAM should be accompanied by a checksum for verification. While a cyclic-redundancy check (CRC) is the safest and therefore recommended method, since the amount of memory that it has to be calculated over can be substantial, depending on the microcontroller performance, a calculation-based CRC implementation may introduce a delay that is considered too long. In cases where a table-based CRC implementation takes up too much code space in the bootloader and is therefore not ideal either, other and more simple means of checksum calculation, for example the TCP/IP method, may be used as well.

Before the bootloader jumps to execute the application, it should always calculate the application's checksum and compare it with the stored one. If there is a mismatch, it should never execute the application and instead stay in the bootloader to allow for another erase/download cycle.

If non-volatile storage such as an EEPROM is used to configure parameters such as the CAN bitrate, the CANopen Node ID, the serial number etc., a checksum error should cause the bootloader to apply safe default values that will allow recovery under normal circumstances.

The bootloader may calculate and verify its own code area checksum as well, even though the consequences of a mismatch are much less clear-cut, and application-specific. It is difficult to think of a scenario where the detection of a bootloader checksum error causing the bootloader to stop executing itself would be the safer option compared to continuing execution, but such an error should be detectable to allow for the node to be replaced when possible.

There are several options to indicate a checksum error, from setting some I/O status over sending a CANopen Emergency message to adding an error entry to the Predefined error field at object 1003_h, if implemented.

[2] CiA 302 DSP Part 3 Version 4.1.0, not specified in detail but mentioned in CiA 302-3.2chapter 4.4

Resource constraints

The bootloader is recommended to be an individual, self-contained piece of firmware that shares no code with the application and must not depend on it for one obvious reason: As previously said, a faulty application should never be able to interfere with the application maintenance mechanism. The application functionality is commonly the main focus of development since it will be running almost all the time. The bootloader on the other side, albeit crucial if needed, will hardly ever be visible to the end user. The execution of the bootloader and the application is mutually exclusive and both occupy different parts of flash memory. There is always the pressure to save cost as well, and to choose the smallest chip that can perform the task at hand. These requirements combined mean that the bootloader should take as little space as possible and leave the most room for the application. With some devices there are even hard limits when a dedicated boot flash area is used, for example 8 KiB in an Atmel AT90CAN128.

RAM requirements are less crucial since all RAM can be used by the bootloader or the application when the retrospective code is executing.

Application execution

When the regular application is developed and tested, this is normally done on its own and with no bootloader present. A running bootloader is by itself a complete application, too, though. It initializes the microcontroller and uses several of its peripherals such as timers, CAN controllers and perhaps other I/O. When the bootloader has determined that it should execute the application, a straight jump would be the fastest approach, however, now the application starts executing from a different state than when it was developed and tested. It has been found that this approach can cause applications to fail for subtle reasons that can be very difficult to track down. They may also not surface right away but only after the device has been deployed in the field and a new application firmware version has become necessary. For this reason, it is highly desirable to start the application from a state that is as close to the reset state as possible. This can be achieved by jumping not directly to the application but setting a keyword pattern at a certain RAM location or register that is excluded from startup initialization,

generating a self-reset and at the following entry to the bootloader code checking this keyword. If set appropriately, the bootloader immediately jumps to the application in a “blank slate” state, before all peripherals for regular execution are initialized.

Forced bootloader and backdoor entries

When the device has a regular application programmed, it is started automatically after each reset, as long as the checksum verification passes. To trigger a switch to the bootloader when the application is running, for example after object dictionary entry [1F51$_h$,1] (Program Control) is written with 00$_h$ to “Stop Program,” the application can write to the same keyword pattern to signal that a forced bootloader execution is requested, and generate a self-reset. When the bootloader executes first and sees this request, it will not automatically execute the application after checksum verification and instead stay in bootloader mode, waiting for commands.

A distinct challenge rises if the loaded application passes checksum verification but is not working properly, e.g. not communicating on the CANopen bus or immediately crashing, removing our method of forced bootloader entry. In this severe case the only remedy is a backdoor entry method. If physical access to the device is possible, this can be an input pin connected to a button or DIP switch which is checked after the bootloader starts up. However, if no physical access is available, the failsafe mechanism still possible is an initial waiting period after reset. This means that after a regular reset and when not detecting a specific keyword pattern request, the bootloader will initialize itself normally and wait for – specific! – incoming CANopen messages. If it sees a request, which could be an SDO read to its own Node ID, it will process the request and stay in bootloader mode. If no specific communication is seen within a certain time, the bootloader will execute the regular start application sequence. The big advantage of this method is that it makes the bootloader always accessible after a power-up reset. The disadvantage is that regular application startup is always delayed by the initial waiting time.

In summary, Figure 2 shows an execution flow example of a secure CANopen bootloader.

CAN driver implementation

In order to implement the polled CAN driver for maximum security and reliability, its initialization should allow as few options as possible. For the CAN bitrate it should either use a fixed, hardcoded configuration or, if there are any configuration options, should allow selecting between few defined and tested options such as:

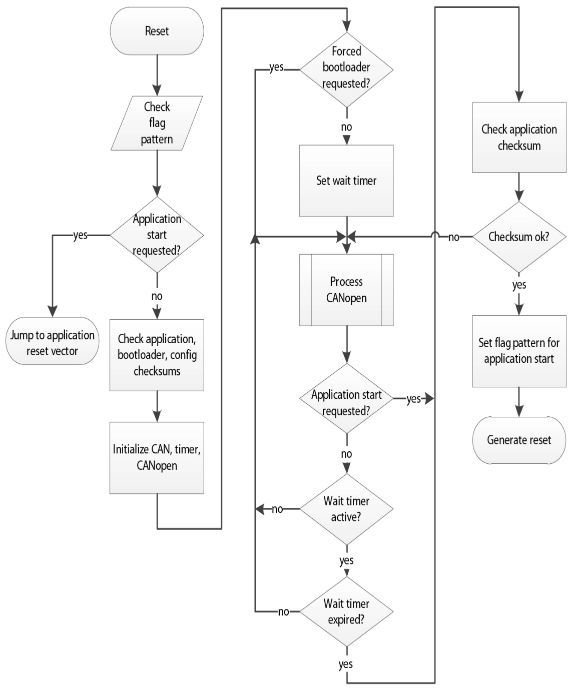
Reset
Check flag pattern
Application start requested?
yes
no
Jump to application reset vector
Check application, bootloader, config checksums
Initialize CAN, timer, CANopen
Forced bootloader requested?
yes
no
Set wait timer
Process CANopen
Application start requested?
yes
no
Wait timer active?
no
yes
Wait timer expired?
no
yes
Check application checksum
Checksum ok?
no
yes
Set flag pattern for application start
Generate reset

Figure 1 – Example execution flow

- Default CAN bitrate
- Alternative CAN bitrate

If possible, CAN hardware message filtering should be used to ignore all CAN traffic that is of no interest to the node and let only two distinct 11-bit messages pass:

- SDO Request (600_h + Node ID)
- NMT message (000_h)

For transmissions, since polling is used, the driver should wait for the transmit buffer to clear either before or after a transmission. Multiple transmit buffers, if available, may be used even though the conceivable benefits will be minimal.

CANopen implementation

The communication requirements of a CANopen bootloader are limited and specific. In addition to a basic NMT state machine and heartbeat generator, there is only a small set of object dictionary entries that has to be implemented in the node, and for all but one the SDO expedited communication method is sufficient. The single exception is object dictionary entry $1F50_h$ (Program data), where its subentries 1 and higher, if applicable, contain a DOMAIN type entry for the application or firmware data. For this type of entry, the SDO segmented or block transfer method has to be implemented. The SDO block transfer may seem like the more attractive option to transfer larger amounts of data with greater data throughput but can be commonly discarded for this purpose because

- it adds a new requirement to the bootloader's polled CAN driver implementation – to be able to receive and buffer back-to-back CAN messages,
- it adds to the RAM and, more severely, ROM requirements of the code significantly, and
- SDO segmented access has to be implemented regardless.

In the majority of cases, the firmware download will be a relatively rare event, and therefore a performance optimization with SDO block transfer will not be motivation enough to justify the increased complexity, limited portability of the code and added potential points-of-failure. Therefore, SDO segmented transfer should be the preferred choice for the DOMAIN entries.

Many services such as all PDO handling, SYNC processing, heartbeat and EMCY consumers are not needed in a CANopen bootloader.

Remote diagnostics

In deeply embedded nodes, all diagnostics of a problem in the node will have to happen via the CANopen network. The mentioned program data entry in the object 1F50$_h$ array can be readable. This is optional, but can be useful for diagnostics to read out the current flash memory contents of the application area. If implemented, its access may need to be guarded with a password against unauthorized access the same way flash programming may be.

To allow access to internal error information, the object 1003$_h$ (Predefined error field) is an obvious choice. It is a dynamic list which means that the number of subindexes can change. Subentry 0 always has the current number of elements in the list to indicate the highest subindex that can currently be read. After reset, and if there was no error, subentry 0 contains 00$_h$ and the list is empty. The latest error is added to the top of the list and the highest available subentry has the oldest error. The entries are 32-bit values with the lower 16 bits containing an error code with many codes already predefined

in CiA 301 and the upper 16 bit holding freely usable additional information about the error:

31 16	15 0
Additional Information	Error Code

Table 1 – Object 1003h, Subentry 1 and up – Standard error field

A possible assignment for bootloader use is suggested below:

Bits 15..0 Error Code	**Meaning**	**Bits 32..16 Additional Information**
0x6100	Internal Software Error: Bootloader checksum wrong	Calculated checksum
0x6200	User Software Error: Application checksum wrong	Calculated checksum
0x6300	Data Set Error: EEPROM checksum wrong	High byte: Calculated checksum byte Low byte: Read checksum byte

Table 2 – Example standard error field entries

Certification

The requirement for some classes of devices to pass certification can impose a challenge to many standard CANopen implementations in the C language, even though the detailed requirements vary widely depending on the standard the code has to adhere to. Since the author has assisted in certification of CANopen bootloader code according to Subsea Instrumentation Interface Standardisation (SIIS) where in turn adherence of the code to MISRA C Guidelines (1998) was required, these shall serve as a non-comprehensive example. Some of the rules that made using pre-existing code impractical and lead to a new coding effort were:

- Type casting from any type to or from pointers shall not be used

This rule was relaxed in MISRA-C-2004 but the original version meant that a generic table-based CANopen object dictionary implementation could not feasibly be adapted to pass that particular check.

- The *continue* and *break* statements shall not be used

These requirements would mean plenty of changes for code that uses those statements, along with significant additional testing.

- All *if…else if* constructs shall be terminated with an *else* clause

This rule is relatively easy to comply with, but can mean a lot of additional "empty" code that also has to be documented.

There are many additional rules in this specific framework that all lead to safer and better maintainable code but may be at odds with pre-existing solutions.

Conclusion

The construction of a bootloader differs significantly from a regular application. It has limited but unique functionality together with constraints in the use of interrupts and code space. This, along with the application of safe coding standards, calls for a dedicated bootloader implementation to ensure maximum operational security and minimize the risk of unrecoverable failures of devices in hard- or impossible-to-reach areas.

References

CiA 301 V4.2.0 – CANopen application layer and communication profile

CiA 302-3 DSP V4.1.0: CANopen additional application layer functions Part 3: Configuration and program download

The Motor Industry Software Reliability Association (MISRA): Guidelines For The Use Of The C Language In Vehicle Based Software, Version 1.0, July 1998

MISRA-C:2004: Guidelines for the use of the C language in critical systems, October 2004

Christian Keydel
Embedded Systems Academy, Inc.
1250 Oakmead Parkway, Suite 210
Sunnyvale, CA 94085, USA
phone +1 (877) 812-6393
fax +1 (877) 812-6382
ckeydel@esacademy.com
www.esacademy.com