CANbedded Program-Stack usage

Version 1.0
2007/01/08

Application Note AN-ISC-8-1056

Author(s)	Andreas Raisch

Restrictions	Customer confidential - subject to prior approval by PSC

Abstract	This application note provides basic information on the (program-) stack consumption of Vector's CANbedded communication components.

Table of Contents

1.0 Overview 1
1.1 Definition “The Stack” 1
1.2 Embedded System Stack Handling 2
1.3 Stack-Usage Approach in CANbedded 3
1.3.1 Example 3
1.3.2 Configuration Aspects 5
1.4 Calculating and Measuring Stack Need 5
2.0 Summary 5
3.0 Contacts 6

1.0 Overview

This application note provides basic information on the (program-) stack consumption of Vector’s CANbedded communication components.

1.1 Definition “The Stack”

In computer science, a call stack is a special stack, which stores information about the active subroutines of a computer program. (The active subroutines are those, which have been called but have not yet completed execution by returning.) This kind of stack is also known as a program stack, execution stack, control stack, function stack or run-time stack and is often abbreviated to just "the stack".

The actual details of the stack in a programming language depend upon the compiler, operating system, and the available instruction set.

As noted above, the primary purpose of the stack is:

- **Storing the return address** – When a subroutine is called, the location of the instruction to return to needs to be saved somewhere.

A stack may serve additional functions, depending on the language, operating system and machine environment. Among them can be:

- **Local data storage** – A subroutine frequently needs memory space for storing the values of local variables, the variables that are known only within the active subroutine and do not retain values after it returns.

Copyright © 2007 - Vector Informatik GmbH

- **Parameter passing** – Subroutines often require that values for parameters must be supplied to them by the code which calls them and it is not uncommon that space for these parameters may be laid out in the stack. Generally if there are only a few small parameters, processor registers will be used to pass the values. The stack works well as a place for these parameters, especially since each call to a subroutine, which will have differing values for parameters, will be given separate space on the stack for those values.
- **Other return state** – Besides the return address, in some environments there may be other machine or software states that need to be restored when a subroutine returns. This might include things like processor registers, privilege level, exception handling information, arithmetic modes and so on. If needed, this may be stored on the stack just as the return address is.

The typical stack is used for the return address, locals, and parameters (known as a call frame). In some environments there may be more or fewer functions assigned to the stack.

A stack is composed of stack frames (sometimes called activation records). Each stack frame corresponds to a call to a subroutine, which has not yet terminated with a return. For example, if a subroutine named DrawLine is currently running, having just been called by a subroutine DrawSquare, the top part of the stack might be laid out like this:

The stack frame at the top of the stack is for the currently executing routine. In the most common approach the stack frame includes space for the local variables of the routine, the return address back to the routine's caller, and the parameter values passed into the routine. The memory locations within a frame are often accessed via a register called the stack pointer, which also serves to indicate the current top of the stack. Alternatively, memory within the frame may be accessed via a separate register, often termed the frame pointer, which typically points to some fixed point in the frame structure, such as the location for the return address.

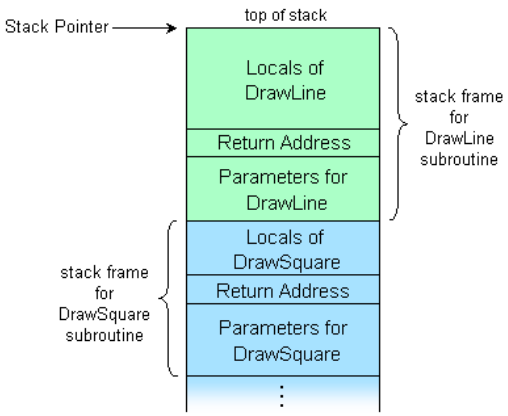
Stack frames are not all of the same size. Different subroutines have differing numbers of parameters, so that part of the stack frame will be different for different subroutines, although usually fixed across all activations of a particular subroutine. Similarly, the amount of space needed for local variables will be different for different subroutines.

[Source: Wikipedia]

1.2 Embedded System Stack Handling

The bandwidth of embedded system architectures and also the used µC (and compilers) results in a huge variety of stack handling approaches.

The simplest approach is often valid, when no operating system (OS) is used. In such a case the stack is simply a



reserved RAM area used by the whole system.

If an OSEK/OS or AUTOSAR OS is used, the stack handling depends on the available features in the OS and the underlying µC. Each OS task has usually its own stack but can share this stack if some preconditions are kept with other tasks. Providing one or more own stacks for interrupts can reduce the stack need of the OS tasks, too.

The stack need of an embedded system depends also on the specific handling for interrupts:

- Storing of CPU registers on the stack or simply switching to a different stack bank
- Supporting/using nested interrupts; i.e. interrupts can cause multiple store actions to the stack
- Configuring own stack(s) for interrupts or using the stack of the currently running OS task (or other interrupt)

Please note that for most implementations the stack pointer starts at a high address and grows to lower addresses.

1.3 Stack-Usage Approach in CANbedded

The CANbedded architecture is optimized for so-called deeply embedded systems with limited RAM and runtime in real-time systems. As a result of this, the architecture is gained to be a good compromise between

- ROM usage (multiple implementation of same behavior vs. in-lining or calling functions)
- RAM usage (static parameters vs. local parameters on the stack; calling a function vs. implementing something short twice, …)
- Runtime consumption (calling a function needs more time than local handling)

A CANbedded function shall therefore

- have only few arguments (none, 1 or 2, very very rarely more than 2 parameters)
- be implemented in a way that local data stored on the stack is minimized
- call no other functions unless it is necessary due to layered architecture

Most CANbedded API functions are implemented in a way that only flags are set which are evaluated on a cyclic function to prevent too deep function call trees (and with this an increased stack need).

An additional task of the CANbedded communication components is the separation of the interrupt driven hardware handling and the task-level view of the application. To be able to react on communication events correctly, parts of the handler implementation are executed on interrupt level and parts are executed on task level.

Which stack and how much stacks are used for the task and the interrupt context depends on the underlying system (see chapter 1.2 for details).

1.3.1 Example

A typical example for such a CANbedded implementation is a transport protocol, where the reaction on incoming events has to be fast whereas the timeout and state handling can be slower. The implementation will therefore be split in a function called within the CAN RX interrupt for fast actions and a cyclic called function for the slower ones. The same is true for the next upper layer, the diagnostic layer. See Figure 2 for details how the diagnostic layer handles the separation of RX event notification in CAN ISR context and further handling on task context.

As a result, the call tree of the RX event handling in ISR context stops latest on diagnostic layer level. All further handling (including application calls) is processed a few milliseconds later from task level with only very few function calls on the stack.

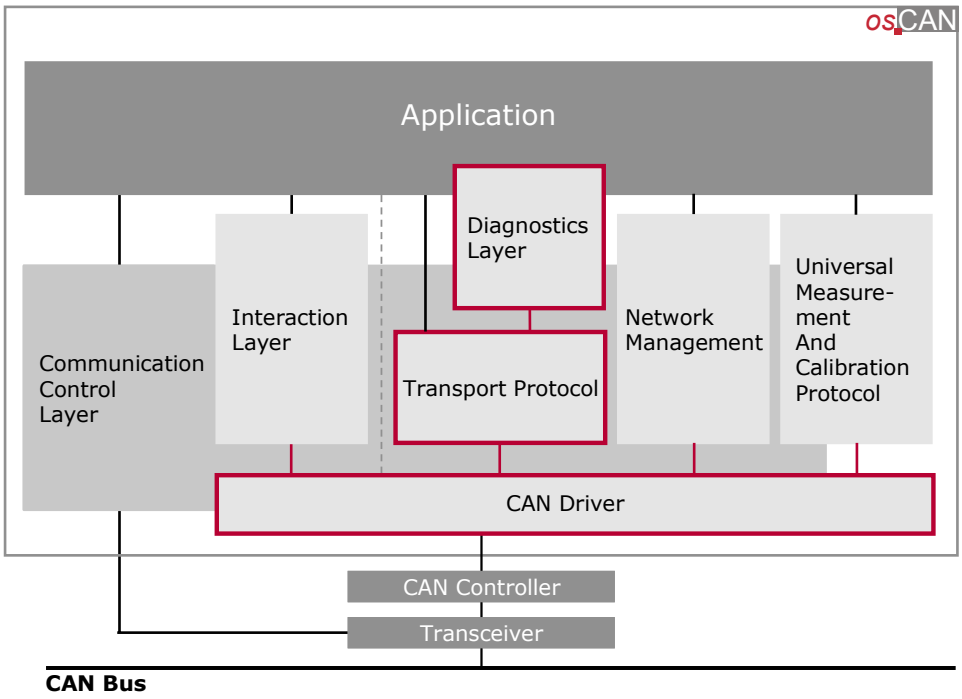


Figure 1 Example CANbedded Stack with CAN Driver, TP and Diagnostic Layer as the largest call-tree

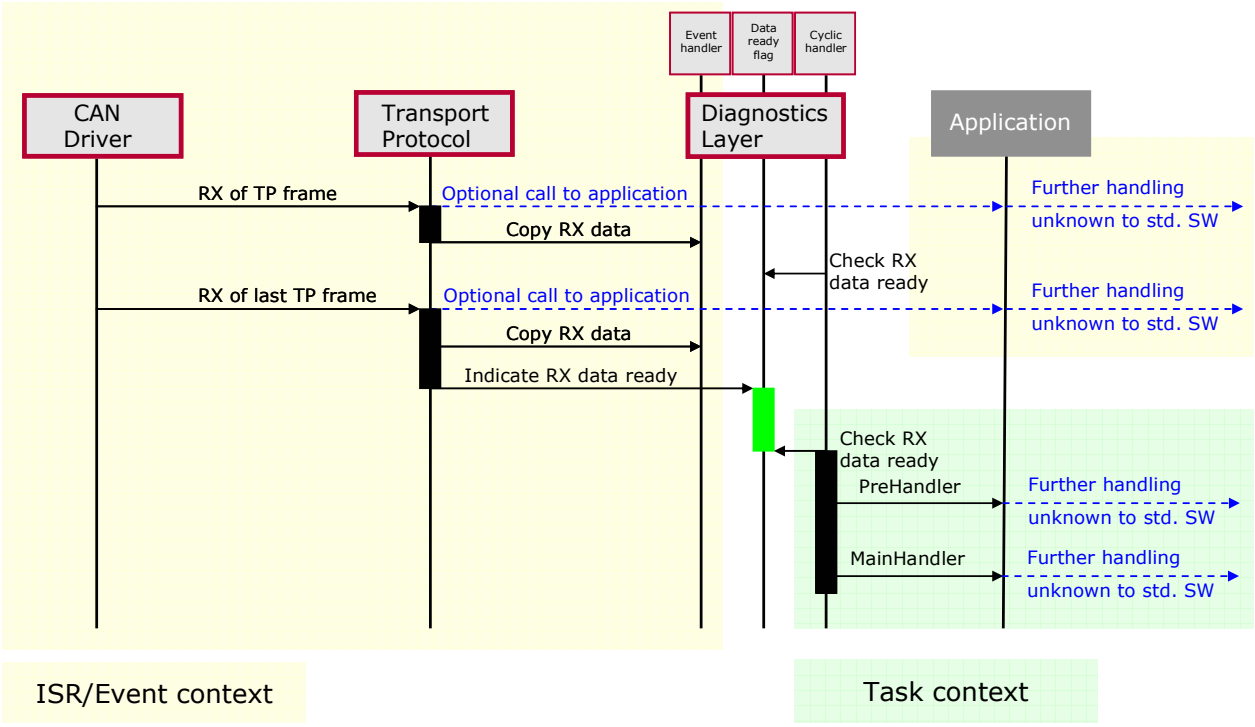


Figure 2 Example call-tree for diagnostic data reception

1.3.2 Configuration Aspects

The configurations of the CANbedded components have a large impact on the stack needs. There is a customer specific, static setup of CANbedded components which interact together to fulfill the OEM requirements. I.e. the CAN driver always informs e.g. the transport layer if a specific CAN frame is received and the transport layer informs the diagnostic layer if the full message is received. This call sequence is preconfigured by Vector, nevertheless, the software supports multiple points (callbacks) where customer specific hooks can be included. This customer hooks have additional influence to the stack needs.

Another very important configuration parameter is the decision if the RX/TX handling shall be processed in interrupt or on task context. If it is processed on task context, the stack need can be calculated easier than when the handling is performed with interrupts.

In an ECU with one CAN bus, an interaction layer (IL), a network management (NM), a transport layer (TP) and a diagnostic component, the RX path of incoming diagnostic CAN messages has usually the largest call tree form communication point of view and therefore the highest stack need.

1.4 Calculating and Measuring Stack Need

Exactly calculating the stack needs is a difficult and error-prone task. Therefore the stack size is often estimated based on the environmental conditions described in chapter 1.2 and a safety value of 10% to 20%, sometimes 100%, is added to the result.

If an operating system is used, this system usually provides means to measure stack usage and test stack consistency. OSEK/OS and AUTOSAR OS for instance provides such means to check the stack with each OS action and also to initialize the stack with patterns so the worst case usage for a given scenario can be found. When using such a method, it is important to analyze upfront the worst case call trees and make sure they have happened during test execution, too. This implies also “special” scenarios like “ECU in diagnostics”, “ECU with high I/O interrupt load”, occurrence of OSEK category 1 and 2 interrupts at the same time, …

2.0 Summary

As a result of all this effects, no detailed figure for the concrete stack usage of the CANbedded communication components can be given upfront of the real usage in an ECU project.

Vector offers different levels of support to customers, starting from telephone hotline for the more simple questions, integrating the CANbedded communication components and other software up to analyzing your ECU setup as a separate project work.

Note:	Please be aware that due to all the points mentioned in this application note Vector can not finally list the stack needs of the CANbedded communication components in your project. The concrete layout and dimension of the stack(s) in an ECU project is the task of the system integrator.

3.0 Contacts

Vector Informatik GmbH
Ingersheimer Straße 24
70499 Stuttgart
Germany
Tel.: +49 711-80670-0
Fax: +49 711-80670-111
Email: info@vector-informatik.de

Vector France SAS
168 Boulevard Camélinat
92240 Malakoff
France
Tel: +33 (0)1 42 31 40 00
Fax: +33 (0)1 42 31 40 09
Email: information@vector-france.fr

Vector CANtech, Inc.
39500 Orchard Hill Pl., Ste 550
Novi, MI 48375
USA
Tel: +1-248-449-9290
Fax: +1-248-449-9704
Email: info@vector-cantech.com

Vector Japan Co. Ltd.
Seafort Square Center Bld. 18F
2-3-12, Higashi-shinagawa,
Shinagawa-ku
J-140-0002 Tokyo
Tel.: +81 3 5769 6970
Fax: +81 3 5769 6975
Email: info@vector-japan.co.jp

VecScan AB
Lindholmspiren 5
402 78 Göteborg
Sweden
Tel: +46 (0)31 764 76 00
Fax: +46 (0)31 764 76 19
Email: info@vecscan.com