Porting Defensive Aid Suite to Vehicle Control System

Jonas Jonsson

Luleå University of Technology
MSc Programmes in Engineering
Computer Science and Engineering
Department of Computer Science and Electrical Engineering
Division of EISLAB
Porting Defensive Aid Suite to Vehicle Control System

Jonas Jonsson

Luleå University of Technology
Dept. of Computer Science and Electrical Engineering
EISLAB

Mar 22th, 2010
This Master’s Thesis describes an initial attempt to reimplement the Defensive Aid Suite functionality, used in the latest version of Combat Vehicle 90 from BAE Systems Hagglunds AB. The current implementation uses an internally developed software platform based on VxWorks from WindRiver. Because this software platform is only used by the Defensive Aid Suite, BAE Systems Hagglunds AB wishes to retire it. The implementation described in this report will be based on the currently used software platform which is based on Rubus OS from Arcticus Systems AB.
The work presented in this thesis was done during fall 2009 at BAE Systems Hägglunds AB, Örnsköldsvik. I would like to thank the department of embedded systems for the great time, Magnus Berglund for the fruitful discussions and especially my supervisor Jimmy Westerlund for aiding me during my work.

Jonas Jonsson
5.5.1 Serial Spy ............................................... 20
5.5.2 Simulated Serial HAL Windows .................... 21

CHAPTER 6 – EVALUATION ....................................... 23
6.1 The Work ..................................................... 23
6.2 Reflection ................................................... 24
6.3 Future Development ......................................... 24
CHAPTER 1

Introduction

1.1 Defensive Aid Suite

Combat Vehicle 90, CV90, is a family of armoured combat vehicles developed by BAE Systems Hägglunds AB. The latest version of CV90, CV9035 MKIII, is equipped with a feature known as Defensive Aid Suite, DAS. It consists of a Laser Warning System, LWS, that aids the crew elude threats which use laser to guide weaponry to its target.

The laser warning system is bought from a subcontractor while the system responsible for countermeasures is developed by Hägglunds.

During initial development of DAS, it became clear that the intended hardware platform lacked support for floating point arithmetics. This made it too slow to be usable. However, suitable hardware was available from a previous project but it was based on a processor which at the time was unsupported by Vehicle Control System, VCS. Therefore, both the software platform known as Extended Control System, ECS and hardware was reused from the previous project in order to cut costs and meet the deadline. Meanwhile, support for this hardware was added to VCS.

1.2 Aim

This thesis aims to reimplement DAS on VCS. The focus should lay on the communication stack used to interact with the LWS. In the current solution this part is provided by the subcontractor.
1.3 Scope

A complete reimplementation of DAS is beyond the scope of this thesis. This work is limited to communication with laser warning system and core countermeasures functionality.
2.1 Defensive Aid Suite

In figure 2.1 an overview of the currently deployed DAS system is shown. It consists of multiple laser sensors connected to a controller. By processing sensor data, the controller can determine the direction and if it is considered a threat or not. When a threat is detected, a message is sent to IM16\(^1\) using a serial communication interface.

What actions to perform when a threat is detected is determined by the IM16. Some actions like storing information about the threat in a log file and

\(^1\)IM16 is the internal name for the micro controller node running the software.
displaying it on the Vehicle Information System, VIS, are always performed, while others are configurable by the crew.

### 2.2 Extended Control System

During development of a previous project, neither hardware nor software supported floating point arithmetics, Ethernet, IP, TCP, UDP or Corba. A new hardware and software platform was therefore developed around a PowerPC processor and VxWorks.

Initially it was intended that ECS would be a more powerful alternative to VCS but as time went more features was added to VCS making ECS redundant. Today, VCS can run on this hardware giving it floating point arithmetics. It also supports Ethernet, IP, TCP and UDP but not Corba since it is no longer in use.

### 2.3 Vehicle Control System, VCS

VCS is a software platform for real-time applications used and developed by Hägglunds. VCS forms the basis for applications that control and supervise functions in the vehicle. These functions ranges from rather simple things such as brake lights to more complex things like the rear ramp on CV90 which involves many conditions, several actors and more than one node.

VCS and the application is built in a monolithic way. This means that VCS and the application is compiled into a single binary. This single binary is then the only code executed on the hardware.

Both applications and VCS follows a set of coding rules designed by Hägglunds but based on MISRA-C:2004\(^2\)[1]. For example, one rule states that all memory must be statically allocated. This reduces the risk of memory leakage which could lead to a situation where the system runs out of memory.

In figure 2.2 an overview of the major layers of VCS is shown. An application is split into three different parts, Common units, Product family units and Product specific units. This split makes it easier to reuse code between different projects. Common units is a general part that can be used in different products, such as a navigation system. Product family units are parts only relevant to a specific product family like the CV90. Product specific units are parts only relevant to a specific variant of a product such as a customer specific communication system. Applications use an Application Programming Interface, API to access features provided by VCS OS.

---

\(^2\)MISRA-C:2004 is a guideline for the use of the C language in critical systems.
VCS OS includes support for commonly used features such as CAN\(^3\), digital and analog I/O, Ethernet, TCP/IP, log files, basic graphic primitives, RS-232, RS-422 and many of the functions usually provided by the standard C library. The standard C library is not used due to potential bugs and undefined behaviour in different implementations. Thread support is provided by Rubus OS which is further described in section 2.3.1.

The basic graphic primitives are used to create an end user interface. This interface can be used by the crew of a vehicle to configure and monitor certain aspects of the system.

Both CAN and basic I/O are not often used directly. Instead an abstraction layer with signals is used. Applications define what signals they use and VCS OS hides the details of reading and writing to I/O pins or creating CAN frames.

A textual user interface, known as Monitor, is used to interact with the application as well as VCS OS itself. It provides menus and commands. A menu is a view of internal information of the system. The view is constantly updated and can be used to monitor a signal, threads or current TCP connections. A command is used to interact with the system. This can be used to change values or turn off certain features. This interface can be extended with specific commands and menus by applications through the API. Monitor is used during the whole life time of the system and is a very valuable tool during development and real world testing.

VCS OS uses a hardware abstraction layer, HAL, to ease porting to other hardware platforms. Currently there are six supported platforms MPC5567, MPC8250, XC167, C167 and a simulated environment supporting both Windows and Linux.

\(^3\)Controller Area Network is a message based protocol, designed primarily for the automotive industries but is used by others as well.
2.3.1 Rubus OS

Rubus OS is a real-time OS from Arcticus Systems AB. It consists of three kernels and basic generic functionality used by them.

The red kernel manages red threads which are strictly time driven. These execute at a predefined periodicity with known execution times. This information is used when all red threads are ordered into an offline created execution schedule. A node can have multiple schedules to switch between, of which only one can be active at a single time. The kernel can optionally measure best, worst and average execution time for threads which can be used to analyse performance.

The green kernel is used to handle external interrupts which is defined with a period time and priority. This information together with execution time of the thread is used when building the schedule for red threads. The green threads can preempt both red and blue threads. Like the red kernel it is possible to measure best, worst and average execution time of the threads.

The blue kernel is used for event-based threads and runs when the red kernel is idle. The threads are defined offline with a specific priority and can be started and stopped at run-time. Two blue threads always exists, the blue idle thread with lowest priority and blue kernel with highest priority. Other blue threads can use any priority between these.

Blue threads support basic synchronization primitives, mutexes and semaphores, signaling, message queues, time based waiting. A blue thread can wait for either of these events to occur and both red and green threads can signal or post messages to a blue thread.

2.3.2 Rubus Integrated Component Environment

Component based development is a way to develop applications by assembling pieces of software units, components, into a complete system. A component is a unit which encapsulates functionality with a defined interface. Rubus Integrated Component Environment, ICE, is a graphical developer tool used to create components or complete systems according to Rubus Component Model version 3, RubusCMv3[2].

This model’s basic building block is called Software Circuit. A circuit consists of an interface and one or more behaviors each with its own entry function. Only one behavior can be active at once and can be changed at runtime. The interface is a description of ports used to interact between circuits. There are two supported types of ports, data ports for data flow and triggering ports for control flow. A circuit receives data through its input data ports and produces output to its output ports.

Multiple circuits can be connected and encapsulated in an assembly or
composition. These are ignored by Rubus when the model is deployed and thus only provides abstraction and an hierarchal structure. The differences between the two is that a composition can be split and parts of it deployed on different nodes. An assembly on the other hand is indivisible and must be deployed on a single node. A composition can be seen as a grouping of related objects and is useful when creating a library of reusable objects.

2.3.3 VCS Config

VCS Config is a graphical tool developed by Hägghunds which is used to create configurations for nodes. What to configure is controlled by something called a baseline. It consists of an XML file describing how the configuration tree looks and behaves in VCS Config. Accompanying it, is a set of Python scripts used to do validations of the configuration and generation of files.

Currently there is only one baseline in use, for VCS OS. But there is ongoing work to incorporate configuration of other parts of the application into a baseline as well. This is a consequence of splitting the application as described in section 2.3. In order for the parts to be reusable there must be an easy way configure them.

The configuration of VCS OS and its HAL are linked in such a way that an application can be compiled for multiple targets at once. If a node needs a serial connection, for example, a serial channel is created in VCS OS. It includes the name, size of buffers and communication related options like baud rate. Each target then links this channel to a serial port. This can be either a real physical port, a simulated one or a mixture thereof as described in section 5.5.2.

The baseline for VCS OS is used to generate Visual Studio Solution files and building Rubus ICE models besides the source and header files with the actual configuration. The solution file contains a project for each configured target with an external compiler, tool chain and building system. Not depending on external tools makes VCS easier to distribute among the developers.
CHAPTER 3

Current implementation

3.1 Overview

In figure 3.1 an overview of the current implementation is depicted. It is based upon a software platform developed by Hägglunds known as Extended Control System, ECS. More information about ECS can be found in section 2.2.

The main application is built using a mixture of C and C++. Most of the

Figure 3.1: Overview of current implementation, grey part is supplied by the sub-contractor.
Current implementation code, which is developed by Hägglunds, uses C++ but the communication stack that is provided by the subcontractor uses C.

3.2 Communication Stack

The provided communication stack is rather advanced with many layers performing specific tasks, such as queueing, calculating checksum or routing. This design provides a somewhat easy way to add, remove or change any of the layers. The existence of a routing layer with support for multiple serial channels suggests that it is supposed to be used in a much larger network rather than a simple point to point connection currently used. Most layers contain statistic information that can be used to supervise the overall health of nodes and the network as a whole.

Documentation on how to use the communication stack was provided along with the actual code. But since it was never intended for Hägglunds to implement this stack on their own, it is missing important details about the protocol.

3.3 LWS Handler

On top of the provided communication stack is a message based protocol used by the LWS. It consists of a number of request and result messages, whereas some result messages can be sent without a prior request. Each message contains a header and data of various length.

The LWS Handler manages this protocol and maintains an overall status of the LWS. Messages related to incoming threats are parsed, stored in a log file and passed along to the Threat Manager.

3.4 Threat Manager

The Threat Manager maintains a priority queue with current threats. These threats can be updated based on new information, removed if it is old, merged into a single threat when two threats are very close, or simply adding a new threat. The threat with the highest priority is passed along to the Defence Manager.

In addition to queuing, the Threat Manager also provides VIS with information used to display an overview of the current threats.
3.5 Defence Manager

The configuration set by the crew affects how the Defence Manager responds to the most prioritized threat. The response is controlled by a state machine, handling possible user interaction and dependencies on other systems like smoke grenade launchers.

3.6 Config Manager

To ease configuration management in other parts of the application, the Config Manager maintains the current configuration. The configuration can be changed in two ways, either by the application itself or by other nodes using CAN.

For example, when a smoke launcher is loaded, the Config Manager receives a CAN message and updates the configuration.
CHAPTER 4

Communication Protocol

4.1 Overview

As there were no documentation about the protocol used for serial communication it was necessary to investigate how the protocol worked. The information provided here is based on the API documentation and source code for the current implementation. The communication is divided in two parts, a generic serial communication bus protocol, described here, and an application specific protocol for the LWS, described in section 5.2. Based on the design described below it seems likely that the generic part is used in many other applications.

Communication with the LWS uses a common serial standard. On this connection, Serial Line Internet Protocol, SLIP[3] is used. It is a very easy protocol to encapsulate a payload of various length on a serial link. It consists of an END marker, decimal number 192, which marks the end of a transmission. Beside this it also includes ways to transmit this END marker in the payload. Historically, this protocol was designed to work with modem connections over plain old telephone lines. These connections were not perfect and suffered from noise which could cause erroneous bytes corrupting the packet. Beginning every message with an END marker would flush any erroneous bytes received during idle time. This method is used by the LWS.

The SLIP payload contains an application specific payload and a frame header. This is shown in figure 4.1.
4.2 Frame Header

At the end of the application payload is a 10 byte frame header which is used by the communication stack. An overview of the message is shown in 4.1 and the frame header is further described below.

**Packing**

Sometimes it is desirable to transmit data stored in data types larger then needed. The API documentation describes such case when a Digital Signal Processor, DSP, stores bytes word aligned. So for each byte, 3 more are wasted. If packing has a value of 8, it means that the stack should only use the least significant byte. A value of 16 means it should use the two least significant bytes and a value of 32 effectively means no packing.

**Idle Request, IDR**

This field is split in two parts, control type and sequence number. The control type is a field of the two most significant bits describing what type
of message this is. A value of 0 or 1 means it is an unacknowledged message, 2 means it is a normal message while 3 means its an acknowledge message. The last 6 bits are used as a sequence number and is only used in normal and acknowledge messages. Every normal message sent is supposed to be acknowledged with an acknowledge message with the same sequence number. If there is no acknowledge received after a certain time, it should be resent with the same sequence number.

**Source Node**

Denotes the node that sent the message, the LWS identifies itself with a value of 1 and the application is supposed to use 0. This is specified in the source code as well as in documents provided by the subcontractor.

**Source Code**

Denotes the part of the node that sent the message. The value 0 is reserved for stack to stack communication while 1-255 is available for the application. Value 0 and 1 is the only values used in this implementation.

**Destination Node**

Denotes the node supposed to receive the message. In the current implementation, this is used to route a message in a network of multiple nodes. The new implementation uses only the same values as in source Node.

**Destination Code**

Denotes the part of the application that is supposed to receive the message. Similar to source Code.

**Frame Check Sequence, FCS**

To reduce the risk of erroneous message a checksum is used. The checksum is calculated using the application payload as well as the previous 8 byte frame header. The checksum is a standard cyclic redundant checksum know as CRC-16-CCITT, it is also used in other protocols, PPP in HDLC-like Framing[4].
The new implementation is divided in a similar way as the current implementation, described in chapter 3, but tries to use Rubus CMv3 described in section 2.3.1.

5.1 Communication Stack

The communication stack consists of two threads, a red and a blue one. The red thread can issue three signals that the blue thread is waiting for. A receive signal if there is data available, a transmit signal in case of ongoing transmission or a timeout signal.

The blue thread is an infinite loop that most of the time is waiting for a signal. The important signals, receive and transmit are described further down. A timeout signal means that a message was sent but no acknowledge was received. The message will be sent again or removed if it has been sent too many times.

5.1.1 Message Reception

Handling of received data begins with the SLIP receive function provided by VCS OS. It will read available data from a serial channel and remove SLIP related information. This is a non-blocking operation since it will not wait for more data if the message is incomplete. Instead it will return a state variable which must be passed as an argument each time the function is called. When a whole message has been received it will be further processed by the stack.

Two important checks are done before the message is parsed. First, the message length is ensured to be large enough for the header. Secondly,
the FCS must be correct. If these are correct, further checks are done to ensure that the source node and destination node are correct. If any of these checks fails the message will be discarded. The packing is only verified for data messages as it was discovered during testing that acknowledge messages had invalid values. It makes sense for the sender to ignore this field since it is useless for the receiver.

When a data message is received, its payload is extracted and queued in the message receive queue. If a normal message is successfully queued, an acknowledge message is queued for transmission. If the queue is full then no acknowledge will be sent and the sender should retry. Section 5.1.3 contains more details about the queues.

When an acknowledge message is received, its sequence number is compared to the last transmitted message. If they are equal, it means that the transmitted message has been received and can be removed. The transmit state described in section 5.1.2 is also updated to reflected the success of transmission.

5.1.2 Message Transmission

There are two types of messages supported by the stack, normal messages and acknowledge messages, each with its own queue. When a normal message is sent, a timeout value is set. If an acknowledgement is not received during that time the message will be sent again a predefined number of times before discarded. Only one normal message can be waiting for an acknowledgement at the same time.

The stack will try to send normal messages before acknowledge messages and it will transmit acknowledge messages while it waits for an acknowledgement of a previously sent normal message.

The SLIP transmit function provided by VCS OS is non-blocking which forces the stack to handle partial transmissions. This is solved with a global transmit state that indicates if a transmission is partially done. The red thread will then signal for continued transmission if needed.

5.1.3 Message Buffering

In order to buffer messages that pass through it, the stack uses three queues. They queue received messages, unsent messages and unsent acknowledge messages. The queues are implemented as circular buffers using a statically allocated array of messages. The queue is realized as a structure containing four members, a boolean flag indicating if it is full, indexes for start and end position and the actual message array. The flag is used to distinguish between an empty or full queue which both is indicated by start and end
being equal. Another way to distinguish would be to only let them be equal when empty. This would in this case waste more memory, since a message is larger than the flag.

5.2 LWS Manager

While the communication stack only serves as a transportation of data, the real work is done by the LWS Manager. It implements the specific protocol transported by the communication stack. It is used to interact with the LWS. This protocol is well documented compared to the one used to implemented the communication stack.

The protocol consists of a number of request and result messages. Each request should be replied to with one result message. Besides this, the LWS can send result messages without a prior request which is important for certain types of messages. Threat detection, threat timeout, status and continuous built in tests all send unsolicited messages. Each message is prepended with a header containing command identifier, sequence number\(^1\), length of payload and time stamp. The sequence number is used to match request and result by the requester. For unsolicited message the sequence number will be zero to indicate that it was not requested.

Parsing of messages are divided in two steps, header and payload. The first step consists of a single circuit that each time tries to receive a message from the communication stack, parse the header, verify the length and write payload and command identifier to output ports.

The second step involves one circuit for each message. Each circuit parses and acts upon the payload supplied with the message. Threat related messages are passed to the Threat Manager described in section 5.3. Messages related to the status of the LWS and its sensors could be used to evaluate if it is functional or not.

5.3 Threat Manager

The two threat related messages received from the LWS Manager are incoming threat and delete threat. The latter is sent by the LWS when the threat is no longer detected while the first is sent when new threat information is available. This could be a new threat or an old threat with updated information.

The Threat Manager maintains a priority queue of active threats. The most prioritized threat is used by the Defence Manager described in section

\(^1\)This should not be confused with the sequence number described in section 4.2.
5.4 while the three most prioritized threat are presented to the crew through VIS.
Besides keeping track of threats the Threat Manager can also merge threats that are close. These merges results in small adjustments of countermeasures in order to improve effectiveness.

5.4 Defence Manager

The Defence Manager handles countermeasures based on information about the threat with highest priority and crew configuration. Due to limited time this part has not been implemented. A suggested implementation can not be discussed in detail without revealing restricted information.

5.5 Other Parts

5.5.1 Serial Spy

\[
\begin{array}{c}
\text{LWS} \\
\text{Serial Spy} \\
\text{DAS Application} \\
\text{Host Computer}
\end{array}
\]

Figure 5.1: Overview of Serial Spy usage.

During the first communication attempts with the LWS it became evident that it would be necessary to spy on the serial communication. It was decided, after failing to find a solution, that an application was to be developed. An overview of the solution is depicted in figure 5.1. It consists of an application that forwards information between two serial devices while at the same time showing it to the user.

The application is built using Python\(^2\) and a module called pySerial\(^3\). The application consists of two threads, one for each device, that reads from

\(^2\)Python is a dynamic programming language, see [http://www.python.org/](http://www.python.org/).
\(^3\)pySerial is a library that encapsulates access to serial ports in an easy way, see [http://pyserial.sourceforge.net/](http://pyserial.sourceforge.net/).
one device and writes to the other. The information passed between the
devices is then parsed and shown. It is shown in color using an ANSI es-
cape sequence\[5\] for easier reading. The default textual console in Windows,
Windows Command Processor, does not understand this. Instead a termi-
nal emulator known as RXVT\[4\] was used. In order to use it on Windows,
Cygwin\[5\] is required.

5.5.2 Simulated Serial HAL Windows

Development has been performed solely in a simulated environment which
works rather well for initial testing. In order to simulate input and output, a
system known as Data Distribution, DD is used. This is basically a defined
way for applications to communicate using shared memory. While this is
good for connecting simulated nodes it is not designed for use with real
hardware. It would be possible to write an application that bridges between
the shared memory and the hardware. This approach was investigated but
it did not turn out to be a good or easy solution. It was decided that the
best solution would be to implemented a driver for the simulated HAL that
uses a real serial device.

The driver uses the WIN32 serial API[6] to interact with the hardware,
two queues to buffer data and two threads, one that sends and another that
receives data.

When an application sends data, it uses the VCS OS API which will buffer
the data before using the Hardware Programming Interface, HPI to start
the transmission. This approach is used on all hardware platforms. This
driver will then buffer it another time and signal to a waiting thread that
will interact with Windows in order to send it.

Reception of data is almost the same but reversed. A Windows thread
waits for incoming data, buffering it and triggers a simulated interrupt. The
interrupt service routine will then use the HPI to buffer the data in VCS
OS. The application will then poll VCS OS for data.

Since the Rubus kernel schedules its threads inside one Windows thread
it was necessary to protecte the data shared between the three Windows
threads as well as between different Rubus threads.

\[4\]RXVT is a virtual terminal emulator for X11, http://rxvt.sourceforge.net/.
CHAPTER 6

Evaluation

6.1 The Work

No one really knew how the current implementation looked like, except that it contained a supplied communication stack and a rapidly developed application. The reason for this lack of knowledge is that it was developed by consultants who has left the company.

Because of this lack of knowledge, the work begun with a couple of days source code browsing. It was then decided to begin with the communication stack. Before this work could begin, an introduction to VCS and VCS Config was needed.

At this point, it became clear that testing would require a real LWS Controller. This created another problem, the code would be running in a simulated environment which had no way to communicate with a real device. It could only communicate with other simulated nodes using DD. After working with DD a couple of days the problem was postponed in order to move forward.

When the basics of the protocol had been found, the implementation of the stack begun. Later that week, a real LWS controller was searched for. It was not really hard to find one. However, it would take six weeks before becoming available for use. This was a real setback but some testing software for the LWS Controller was found. This software looked promising since it appeared to contain a simulated LWS Controller. Despite looking good, the most of the software proved rather useless. It was undocumented and did not communicate using the serial protocol. Instead, it used some TCP based protocol. Even though the application payload looked to be correct, it was not useful for testing the serial communication.

Almost three weeks after the first search, an older LWS Controller was
found. Custom cables needed to be made and a day later the testing was to begin but the postponed problem with the mixture of simulated and real environment remained. The solution was to create a driver for the HAL used by the simulated environment connecting it to a real serial device. This implementation is described in section 5.5.2.

When the communication was up it did not really work and debugging was hard mainly because there was no way to inspect the traffic. A tool, Serial Spy, was developed to inspect the serial communication. This tool is described in section 5.5.1. With this tool the bug was easy to spot. It was a faulty assumption about the protocol made earlier during the initial investigation.

Now that the basic building block was constructed, work continued with the LWS Manager and Threat Manager. These proved to be rather easy to implement and focus shifted towards the Defence Manager. It turned out to be more complex than first anticipated and was not investigated further.

6.2 Reflection

A lot of time was spent on analysing how the current implementation worked which influenced the new implementation. Despite the fact that the two implementations use rather different design approaches, object oriented versus component object model, they share the big overall structure. A completely new design would probably been better.

A rather complex system like DAS really needs some prior experience with Rubus CMv3 in order to make the best use of it.

While not accomplishing a complete implementation of DAS, the work that has been done will certainly help. Mostly because the major part that was provided by the subcontractor, the communication stack, is reimplemented, documented and reduced in size. The stack is about one tenth in size. This is accomplished by removing features that are useless such as routing, statistics and compile time layer configuration.

6.3 Future Development

A future implementation should extend the communication stack with a monitor menu covering some basic statistics such as how many discarded messages and how many messages that have been received. It also needs more testing with a higher load. No sensors were ever used during testing due to limited time and availability of hardware.

The new implementation of the LWS Manger only deals with some basic messages and does not handle diagnostic messages sent by the LWS. This
part should be expanded and interact with the diagnostic system used in the vehicle.
REFERENCES


