A new architecture for automotive CPUs

A proposal for secure high-performance computing in cars

Dr. Ken Tindell, CTO Canis Automotive Labs
ken.tindell@canislabs.com

Document number	1902
Version	04
Issue date	2019-11-26

1 Introduction

1.1 A move to central domain controllers

In the 1990s the automotive industry moved to a 'multiplex' architecture for in-vehicle electronics: the number of wires carrying individual sensor data between electronic control units (ECUs) were growing quickly and if unchecked there would have been a huge rise in weight and cost. These many signal wires were replaced with a bus – Controller Area Network (CAN) became the standard – to carry the sensor data in real-time messages.

This move posed a number of challenges for the industry, the largest being cultural: the multiplexed bus represented a resource shared across all the divisions of a car company and access to this shared resource required a centralised group within a car maker take responsibility for integration, ensuring all the ECUs across different divisions of the car maker were able to operate correctly with the specific communications demands.

The biggest integration issue in a multiplex system is one of message latencies: with individual signal wires the latency is essentially the speed of light. On a shared CAN bus the time taken for a message is non-trivial: the CAN frame is generated, queued in a CAN controller, a number of arbitration rounds take place, the frame eventually wins arbitration, is transmitted on the bus and arrives at every other CAN controller. The developer of any individual ECU cannot know what the worst-case latency will be: it depends on what other CAN frames there will be on the bus. Only a group with an overall view of all the bus traffic can do any kind of analysis on this and determine if the system is going to work.

The distributed control system approach has been very successful over the last 25 years and in-vehicle electronics has grown hugely in size and complexity. It is now not unusual to have half a dozen or so CAN buses in a vehicle connecting more than a hundred ECUs. But this is clearly reaching a limit just as the growth in individual signal wires was reaching a limit 25 years ago when multiplex was introduced. This coincides with the widespread adoption of advanced driver-assistance systems (ADAS) that use high-bandwidth sensors (typically multiple high-definition cameras and lidar imaging) which requires high-speed networking, driving a move to Ethernet.

The industry is now moving to a new concept: a few powerful 'domain controllers', running software to implement the functions of the many ECUs, with simple remote I/O devices connected via CAN to provide sensor data to and accept actuator from the domain controllers.

One of the goals for the domain controller architecture is that the supply chain will change its structure from traditional supply of complete mechatronic systems to supply of CAN-connected smart sensor and actuator units plus supply of intellectual property in the form of re-usable software components that run on the domain controllers. These components may come from external suppliers or from

development groups inside car makers as part of a concurrent trend of insourcing domain expertise.

Just as multiplex introduced new challenges in the 1990s, the 2020s will have new challenges. This document sets out the challenges around software components and proposes a solution in the form of a new CPU architecture.

1.2 Embedded software components

The automotive industry today has a well-developed strategy of software components. The AUTOSAR standards were conceived in part to foster the creation of functional software components by defining standard APIs to in-vehicle computing platforms.

The basic element of a software component in AUTOSAR is a *runnable*: a single thread of execution to perform some function. A real-time OS executes collections of runnables within a priority pre-emptive multitasking system that collectively perform functions.

Runnables communicate using the concept of *signals*: small elements of data (typically ranging from a few bits to 32-bits) that are produced by one component (the publisher) and consumed by other components (subscribers). This directly maps on to the communications concepts introduced by multiplexing (in effect a virtualisation of pre-multiplex signal wires). AUTOSAR defines an environment where signals are transparently mapped from one component to another in the same ECU and across in-vehicle networks.

Software components of runnables and signals will form the bulk of the software that runs on a domain controller. These software components will often be supplied to the integrator of the domain controller as pre-compiled (binary) modules. This is for two reasons:

- Binary code is the only form where the supplier can make guarantees of correctness: tested for code coverage, worst-case execution time, etc.
- Guarding intellectual property rights. The source code would represent to many suppliers an unacceptable risk of disclosing their trade secrets (although this is perhaps an unwarranted stance now that there are tools like Ghidra able to quickly reverse-engineer firmware back to source code).

A single computer running many software components from diverse suppliers poses a new challenge: how can a high-integrity secure system be built this way?

1.3 Challenges of high integrity and security

Some software components will inevitably contain defects. Those defects must be confined to the failing software component and not be allowed to propagate. There are two main reasons for this:

- Fault confinement is an essential part of any high-integrity system. If a fault can propagate from one component to another then it is not possible to use evidence of the reliability of that component in isolation.

- A component can be correct but exhibit a fault induced by another failed component. The supplier of a component cannot be expected to assume the liability for the component if its behaviour is out of their control. On the other hand, the integrator cannot assume liability for the whole system without the ability to determine culpability.

There is a further reason for requiring that components are isolated: security. A component with a programming defect could contain a security vulnerability. For example, a carefully crafted diagnostics message could cause a buffer to overrun and the message payload could overwrite a specific data and give the attacker control. If a compromised component can reach into other components, then the attacker can essentially take control of the whole domain controller – and therefore the actuators it controls. This is obviously a catastrophic outcome, especially if the attacker can launch the attack remotely from outside the vehicle.

The car maker obviously has no control of the security of a domain controller – or the vehicle – if a defect in one software component could compromise the entire system. When there are potentially thousands of individual software components in one system it is unrealistic that every single component is free of vulnerabilities.

The timing behaviour of a component has security implications. If a software component executes for too long, then it can cause other components to miss deadlines and fail. In normal systems this is a denial-of-service attack, but in a real-time system it can be used to cause specific deadlines to be missed which have specific implications for the physical system (e.g. failing to move an actuator on time). There are also timing failures that can leak information across security boundaries within a CPU (for example, attacks exploiting branch prediction and cache timing effects).

These challenges mean that the system must be composable: the resource usage specification of individual components must be relied upon. This leads to the requirement for *enforcement* of these properties so that a failure in a component can be isolated to that component. Specifically, there are two types of resource enforcement:

- Timing. The response time of a runnable depends on what other runnables are using the CPU, in much the same way that the latency of a CAN frame depends on what other traffic there is on the bus. The CPU time taken by a runnable depends on how often the runnable is triggered and for how long it executes when triggered.
- Memory. A failed software component must not be allowed to corrupt the memory of another component. This obviously includes overwriting random parts of memory, but also includes dynamic memory allocation (for example, the stack usage must be bounded).

The real-time behaviour of each runnable in a complete system can be determined using timing analysis (just as worst-case latencies can be calculated for CAN frames). But the analysis relies on specific timing properties of each runnable. For the overall analysis to hold then the timing properties must be enforced. To do this properly requires hardware support, and that requires a new CPU architecture.

1.4 History repeating

CPU architectures today are not well-suited to the automotive industry's new domain controller concept. This is no surprise: designs today are the result of a long development road that started from requirements that no longer hold. This has happened before. Through the 1960s and 1970s CPUs gained more and more instructions: microcoding meant it was relatively easy to add exotic instructions by just changing the microcode ROM. The VAX CPU of the early 1980s even had instructions for string handling. In the mid-1980s it was realised that these instructions were hardly ever used – because programs had become written in high-level languages and compilers did not generate them. This led to the RISC paradigm: use a simple CPU design that executes very quickly the few instructions the compiler does use, leaving complexities such pipeline bubbles to be dealt with by the compiler. The result was an explosion in new CPU designs and a step change in performance.

Once the initial performance gains of RISC had been achieved, further gains came from two approaches: more sophisticated fetch-execute designs (out-of-order execution, branch prediction, etc.) and multiple CPU cores per chip. But there are increasing problems with these strategies:

- Complex CPUs execution units have variable timing behaviour. This is bad for security (as mentioned earlier) but also for embedded systems where lack of determinism is a problem: embedded systems must complete computation by a deadline determined by real world behaviour (e.g. the rotation of an engine crank, the movement of a hydraulic brake piston). Running quickly only most of the time is not helpful.
- Multiple cores accessing memory results in bus contention. This results in highly variable execution time as accesses are stalled by bus arbitration logic. Adding caching for performance causes further problems of cache coherency and variable execution times due to cache behaviour when there are interrupts. Again, running quickly only most of the time is not helpful.

CPU design has been driven by the desktop computing paradigm: various applications are running with an OS that schedules them. Neither the OS nor the hardware know anything about the applications and execution is on a best-effort basis. It is faintly ridiculous that phones, printers, cameras and TVs today mostly run on an OS that is a flavour of Unix, designed in the 1970s as an OS for minicomputers with tape drives and drum disks.

There is a different paradigm for computing that, like RISC, also had its origins in the 1980s: the Transputer. This was a CPU designed for massively parallel systems. Instead of the shared memory bus of today's multi-core systems it had closely coupled memory and communications links to neighbouring processors. A language was developed for writing applications – Occam – and a formal method for describing parallel systems: Communicating Sequential Processes (CSP). The Transputer approach failed for many reasons, but one of them was the requirement to break an application into many communicating processes in order to take advantage of the parallelism: RISC designs based on speeding up single-

threaded execution were advancing quickly and did not require software to be re-written.

Automotive software today is already composed from many communicating processes: runnables communicate with each other within an ECU and between ECUs. The new CPU architecture proposed here adopt some of the ideas pioneered by the Transputer.

2 Massively Multi-Core Computing

2.1 Introduction

The new CPU architecture is based on a grid of many CPU cores with tightly coupled memory.

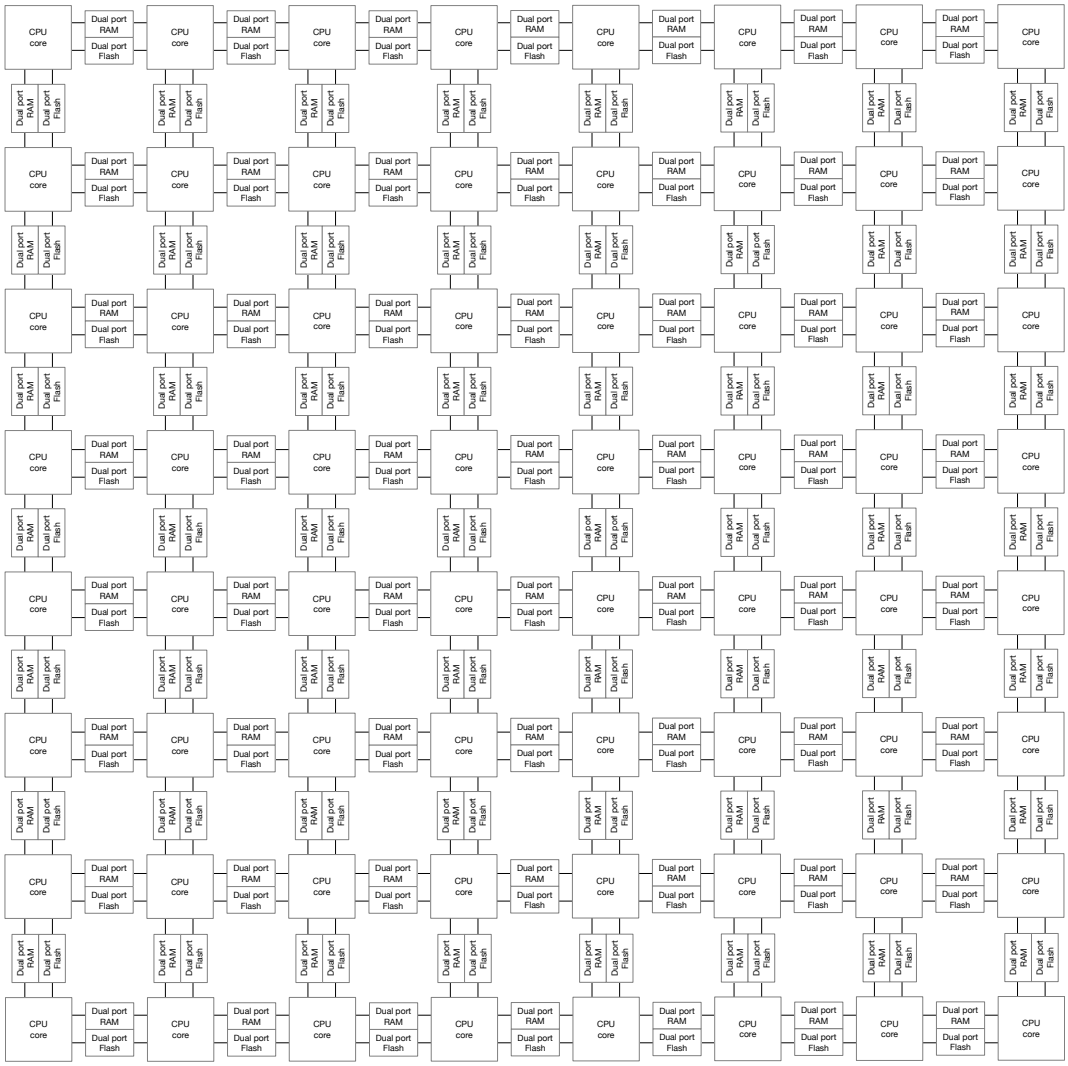

Figure 1: Massively multicore architecture (64 cores in the example)

The tightly coupled memory avoids the need for a shared memory bus, and so avoids the problems of unpredictable timing behaviour when accessing the memory. The specifics of the memory architecture are discussed in the next section.

A common metric for comparing the speed of CPU cores is DMIPS/MHz. But for a massively multi-core CPU the key metric is MIPS/MHz/mm²: this dictates the raw performance of for a given silicon area. The table below gives data for three different Arm cores:

Core	Floorplan/mm²	DMIPS/MHz	CoreMark/MHz	DMIPS/MHz/mm²
Cortex M0+	0.0066	0.9500	2.4600	143.9394
Cortex M3	0.0200	1.2500	3.3400	62.5000
Cortex M7	0.1050	2.1400	5.0100	20.3810

Table 1: Relative CPU core performance for floorplan area

The area values in the above table are for 40LP (a 40nm process). The table shows how a Cortex M0+ is the most efficient way of obtaining raw processing power from a given area of silicon. For example, 0.45mm² could contain 68 Cortex M0+ cores or 4 Cortex M7 cores. The total raw performance with the Cortex M0+ would be 64.6 DMIPS/MHz vs. 9.17 DMIPS/MHz for the Cortex M7.

Obviously, raw performance is only obtained if the large number of simpler cores can be highly utilised. This can be achieved for automotive systems by using allocation optimization techniques (discussed later). Note that there is no requirement for the grid to use homogenous cores: it would be possible to design a device with some of the cores having high performance to handle individual runnables which could not run fast enough even if exclusively allocated a standard core.

2.2 Closely coupled memory architecture

The diagram below shows the structure of the memory between each pair of CPU cores.

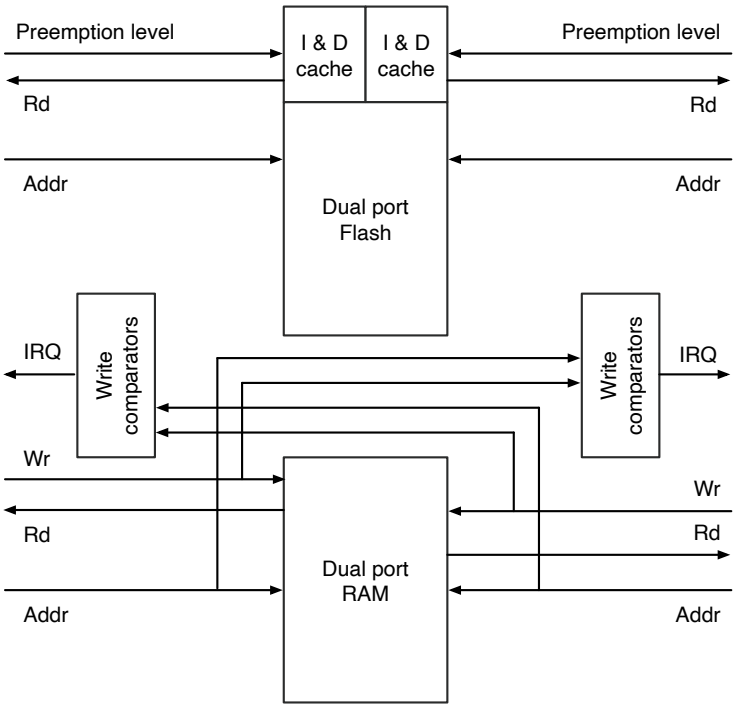

Figure 2: Dual-port RAM and flash memory structure

The RAM is static dual port RAM: each core can write to the RAM without stalling (concurrent writes to the same cell can be avoided by the way the RAM is allocated and the software configured). The address lines accessing memory are passed to a

comparator that can raise an interrupt if certain data is written to certain addresses (the comparator operates on must-match/don't-care pairs to allow ranges of address and data). This allows inter-core communication events: data can be written to an area of RAM and then a write to a specific address used to indicate that the data is present. This allows inter-core message passing to be implemented: messages can then be passed in multiple hops from core to core (typically via real-time tasks with bounded latencies – this is discussed later).

This architecture supports the signals communication model of automotive software components: signals are written to bit positions within a memory area and these are read by the receiving core. They may then be 'gatewayed' to more cores. Concurrent writes to all four RAM blocks are possible by mapping the RAM into 16 address space segments and using the upper four bits of the address space to select to which of the four RAM blocks writes are directed. Publishing signals to neighbouring cores is very efficient, allowing a global signal to propagate through the grid very quickly.

This maps directly on to today's implementation of communications over CAN for software components.

Flash memory is also shared between cores. It is too slow to be accessed directly – a CPU would be stalled with wait states waiting for the flash read (typically 6 or more) and cache on both the instruction and data paths is provided (this is an approach used today for most Arm-based microcontrollers). Typically, caches are interfaced to a pre-fetch module that maps very wide flash words on to the 32-bit words the CPU reads.

The caches are each partitioned four ways and indexed by a 2-bit CPU pre-emption level signal from each CPU core. This is to eliminate the unpredictable execution time that comes from caches being disrupted by pre-emption (such as interrupt handlers running then returning). This is described in more detail later when discussing the scheduling model of the CPU cores.

The flash memory is also dual-ported: it can be accessed by both cores at the same time. This is designed to allow sharing of common code but also to give the greatest flexibility to the component allocation algorithm.

2.3 Flexible memory allocation

Each core can see four flash blocks and four RAM blocks. A core that has components which need a lot of flash for tables or for code can be allocated most of the four flash memory blocks. Common code (such as shared libraries) can be accessed from both cores.

The publish-subscribe communications model for software components means that a signal could be generated by a distant core and be hopped across cores (by being copied from core to core) to software components that subscribe to the signal. The signal need only be stored once in a dual-port RAM block to be accessed by software components on both cores.

The adjacent cores can still see other memories, of course, as illustrated below.

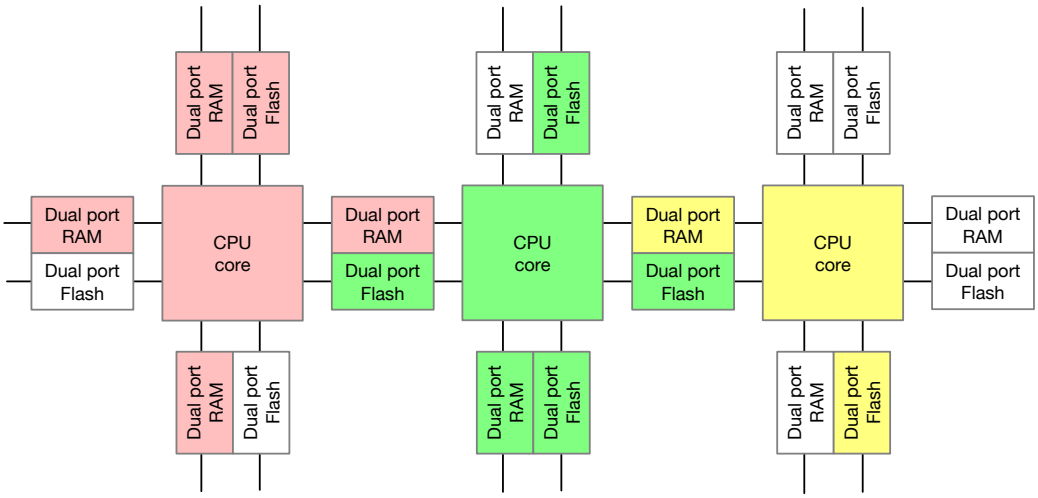

Figure 3: Flexible memory allocation between cores

In the above example, the green core is given all the flash memory blocks because it contains components that need a lot of code or constant table space. The yellow core needs only a little so is allocated memory in just one flash memory block. The allocation algorithm also allocates RAM in a similarly flexible way.

This dual-port approach to closely coupled memory means that the overall use of memory can be very efficient: the allocation algorithm has a lot of freedom to allocate the components and the memory usage across the grid of cores.

2.4 Clocks

There is no requirement for a common clock for all cores. Each core can have its own clock domain, and data can cross clock domains using dual-port RAM. This avoids the clock distribution problems associated with maintaining a single clock across a large area. This allows each core to be clocked faster than if there were a single global clock. A typical implementation would allocate a single clock PLL to a local group of cores.

There is a need to schedule runnables on different cores at the same time where the outputs must meet a specific global time requirement (for example, where four actuators must be moved at the same time). This can be provided by having a slow global time clock (say 1MHz) to allow for timestamping and scheduling of events within each core.

2.5 I/O

In the new grid architecture, most I/O is external to the device and accessed via a communications link. Cores at the edge of the grid are interfaced to communications controllers. Typically, these are CAN and Ethernet MAC controllers, interfaced via pin multiplexers that allows controllers to be spread across a few cores or concentrated into one.

The architecture is designed for flexibility: to allow one or more cores to be used as communications processors (implementing a high-speed switch) or to allocate communications-intensive software components on nearby cores for short latencies to and from the communications cores.

The new CAN-HG augmentation of CAN developed by Canis Automotive Labs allows for large payloads (up to 96 bytes) at high speed (10Mbit/sec) with the same latencies as existing CAN networks. Sensor and actuator modules could use a zero-software design of a small CAN-HG to SPI gateway device that simply relays SPI transactions: all device drivers would reside in the domain controller and send SPI transactions at high speed in CAN-HG payloads. This arrangement is robust (CAN-HG has several features to protect payloads from noise) and secure: CAN-HG includes anti-spoofing and other security protections in hardware[1].

[1] See Canis Automotive Labs document 1901 "CAN Bus Security**:** Attacks on CAN bus and their mitigations" for a detailed description of CAN bus security and hardware mitigations.

3 CPU core features

3.1 Introduction

The new architecture includes novel features in the individual CPU core. These are to provide a secure platform for running diverse software components so that a fault in a component (either a genuine error or an exploited security vulnerability) cannot propagate to unrelated software components. One of the consequences of this is that CPU core features are designed to optimize the worst-case performance rather than the average case.

The features added to a CPU core are:

- Basic real-time operating systems (RTOS) features for multi-tasking implemented in hardware.
- CPU time demands of software bounded and enforced in hardware: overruns are detected so that unrelated software will not fail.
- Memory protection so that a bug or an exploited vulnerability in one software component cannot cause another to fail.
- Hardware vector tables for secure access to runnables and signals.

Priority pre-emptive multitasking is a common feature implemented in many existing RTOSes as well as the AUTOSAR framework. The AUTOSAR model builds on this by defining *runnables* as components that are allocated to run inside tasks. For example, a task that is activated every 10ms might execute a sequence of runnables – implemented as function calls – inside the body of the task. This is a very efficient way to execute software components.

The new CPU architecture supports the above directly in hardware. Prioritized interrupt service routines (ISRs) execute almost exactly like the tasks described above and this is the basis of the hardware support for tasking.

3.2 Multi-tasking in hardware

The new CPU architecture defines interrupts as having a *base priority* and a *boost priority*. When an interrupt becomes pending its base priority is compared against the current execution priority and, if higher, the ISR for the newly raised interrupt is run with the current priority set to its boost priority.

Runnables are called as functions from the ISR. Each runnable has only a boost priority and the current priority is raised to this when the runnable starts and restored when finished. The concept of AUTOSAR runnables is extended to allow a runnable to call another runnable in a nested manner with the priority boosted around each call. This is provided to control concurrent access to a shared resource, and is a concept used by AUTOSAR and other systems (for example, protected records provided by the Ada programming language).

The boost priority of a runnable is calculated at build time and set to the *ceiling priority*: the highest base priority of any ISR that can call the runnable (either directly or indirectly via another runnable).

The diagram below gives an example execution to show how priorities are implemented:

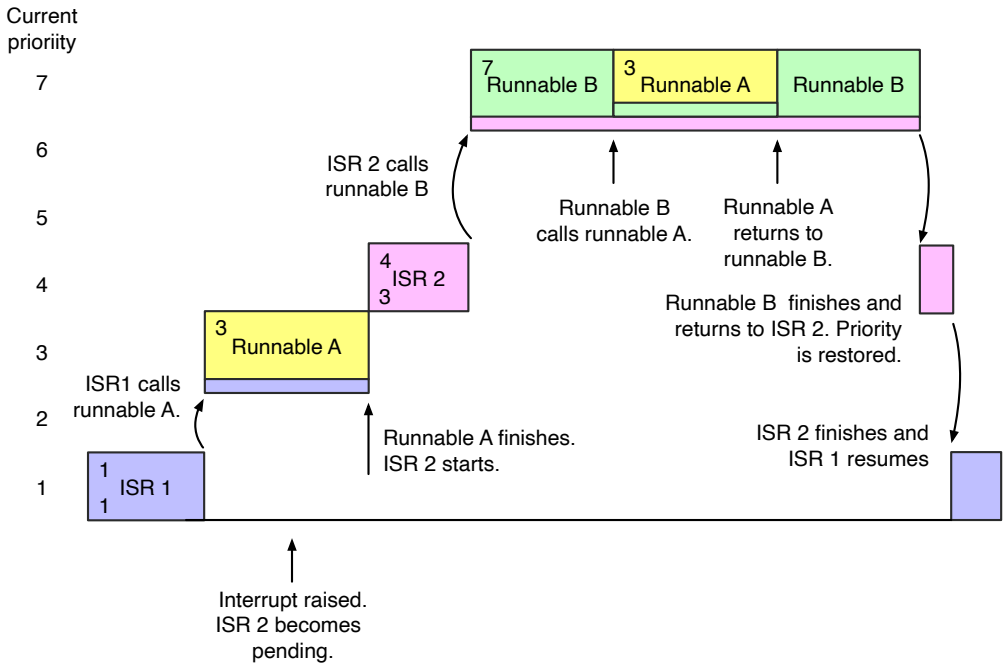

Figure 4: Example of priority pre-emptive execution across two interrupts

In the above diagram ISR 2 calls runnable A indirectly (because it calls runnable B) but is prevented from interrupting an existing call of runnable A until after the call finishes.

The reason for the priority boosting of ISRs is to restrict pre-emption (i.e. prevent a higher priority ISR from running). This is done to reduce the worst-case resource usage:

- Pre-emption costs CPU time to perform the saving and restoring of CPU registers between ISRs.
- Pre-emption costs stack space: somewhere to save and restore the registers, and the stack space used by the application when a pre-emption takes place.

The most efficient real-time system is where there is no pre-emption and the complete application runs at one level. Priorities are needed to arbitrate between different urgencies but once arbitration is decided, pre-emption may not be necessary. This is exactly how CAN bus works: priorities are used for arbitration at bus idle but once arbitration is won, a CAN frame is transmitted without pre-emption.

Pre-emption is needed only if deadlines cannot be met without it. For example, if one runnable has a long execution time and would delay an ISR calling an urgent runnable then pre-emption is necessary. In many cases a runnable with a long

execution time can be restructured to be a sequence of shorter calls that effectively yield a task switch point in between each shorter call (this is why CAN bus frames are limited to just 8 bytes: the most urgent frame can be delayed by at most one short frame taking about 250μs). In Figure 4 there are multiple nested calls and priority changes but only two pre-emption levels.

The number of pre-emption levels required depends on the specific timing requirements and execution times of the tasks. There are algorithms to assign boost priorities to reduce pre-emption and the results of applying these to typical task sets shows that there is rarely a need for more than four pre-emption levels.

3.3 Timing analysis

Since the 1980s there has been extensive research into *response time analysis* that calculates the worst-case response time of tasks in a system that are dispatched according to priorities. These worst-case response times are compared against static deadlines to determine if the system is schedulable.

If the timing analysis indicates that a runnable will meet its deadline then, in all valid permutations of executions of ISRs, the runnable will complete on time. This is an essential property when integrating diverse software components into a single core. But for the results of timing analysis to hold at run-time the attributes fed into the timing analysis must hold at run-time. These attributes include:

- How often each ISR runs
- The execution time of each ISR and which runnables it calls
- Base and boost priorities of each ISR
- Which runnables are called by other runnables
- The execution time of each runnable

By enforcing these attributes in hardware, it becomes possible to make absolute guarantees of timing performance: failure to adhere to an attribute will cause a fault in that software component but the fault cannot propagate to unrelated components. These enforcement mechanisms are part of the CPU core.

3.4 Secure signalling

Signals are the primary mechanism for communication between software components. When communicated in CAN frames, they are packed for efficiency. To reduce dual-port RAM use they can also be packed in memory. Consequently, accessing bitfields of a word is a very common function in automotive software component communication. It can be quite slow when done in software, particularly if the CPU does not have a barrel shifter. For security, writing to signals must be restricted to the authorized publisher of a signal value. Only authorized subscribers should see signal values.

Both efficient and secure signalling is accomplished through a *signal vector table* (SVT). Each entry in the table contains a pointer to the dual-port RAM where the signal value will reside, plus attributes of the signal (for example, how many bits, the offset within a word, whether the signal is published or subscribed). The CPU

detects a write to the SVT and converts it into an access to the specific bits within a word in the RAM. Similarly, a read from the SVT will extract the appropriate bits from the word in RAM. The SVT base and limit registers can be set independently for different runnables so that access to signals can be restricted to only the runnables that need the access.

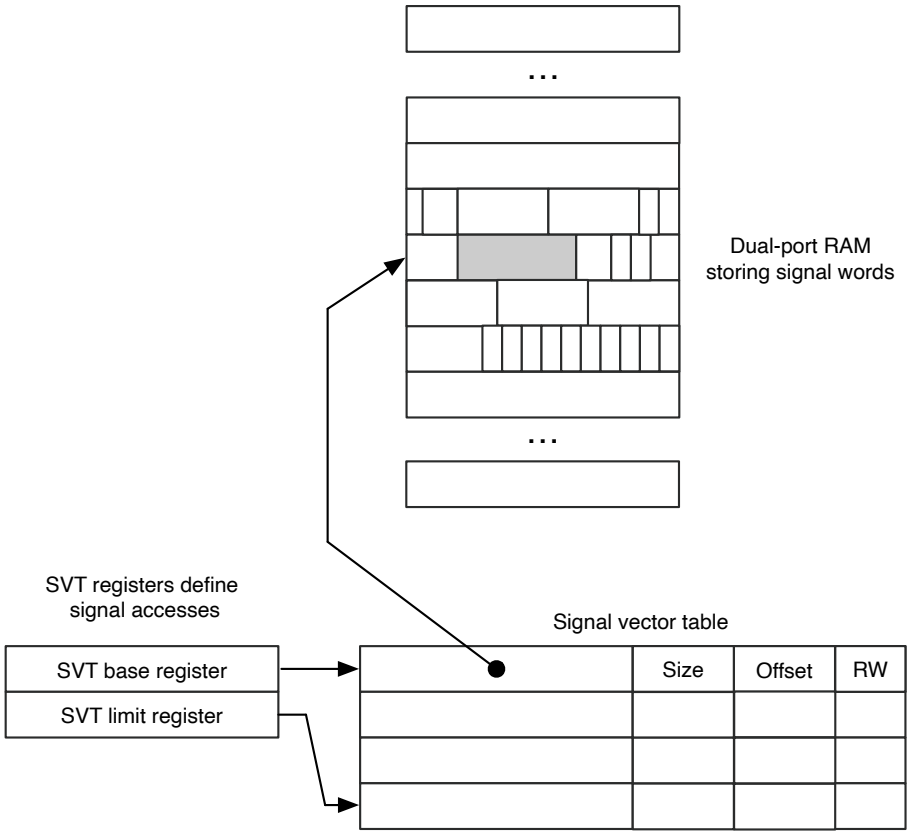

Figure 5: Accesses to signals via the Signal Vector Table

The above diagram shows how signals are accessed by a read or write to an address within the signal vector table. Changing the SVT registers when a runnable is called allows the permitted signal accesses to be set for the new runnable.

3.5 Rate limiter hardware

The rate at which a given interrupt source is raised must be protected. Otherwise a badly-programmed timer or network interface could cause the CPU to become overloaded with interrupts – and the assumptions of the timing analysis used to integrate software components will have been violated. To protect against this each interrupt source has rate limiter hardware.

The characterisation of an interrupt needs to balance two factors:

- It must be simple and require modest hardware to implement.
- It must reasonably accurately model the real behaviour so that the timing analysis is not too pessimistic.

Pessimism is a general problem with any kind of analytical approach to engineering: if the assumptions are wildly pessimistic then the analysis will too

often indicate a system is infeasible when it is not. For example, consider a system where periodic message of 10 bytes is sent every 100ms over a 9600 baud serial line where there is 1.1ms between each byte and it takes 100µs of CPU time to handle each byte. The long-term CPU load is 1%. A simplistic analysis that assumed an interrupt every 1.1ms would assume a long-term CPU load of 9%. This would lead to hugely under-utilised CPU cores.

The rate limiter model of the new architecture has the following attributes:

- The period. This is the minimum time between events that give rise to the interrupt (sporadic events must have a minimum time).
- The jitter. This is the maximum variability between the event source and when the event is raised in the hardware. This typically happens when there is a variable delay from event source to interrupt raise (e.g. a message across a network with a bounded but variable delay).
- The maximum number of interrupts in a burst associated with the event. This allows a single event to cause several interrupts (e.g. a periodic message received on a serial line and where each byte causes an interrupt).
- The burst window. This is the time in which all the interrupt raises in a burst must occur.

The timeline diagram below illustrates the characterisation:

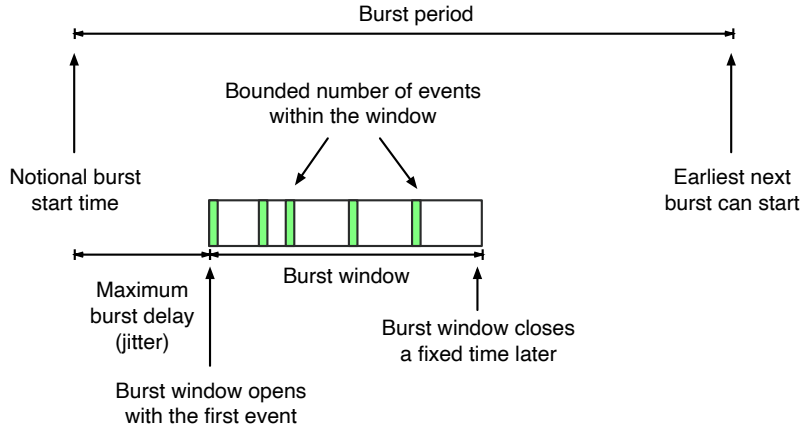

Figure 6: Characterisation of events handled by rate limiter hardware

Any raises of an interrupt that fall outside of rate limit are ignored (an error can be flagged).

This characterisation of an interrupt is compatible with existing response time analysis without excessive pessimism. It also can be implemented in hardware with relatively few gates.

Every interrupt source is assigned a rate limiter but there are also rate limiters that can be used in software to limit the rate at which runnables can be called. This is useful when an ISR is shared between several hardware devices and invoked when any requires servicing. The ISR calls various runnables, depending on which hardware devices have flagged an event (typically the runnables are device drivers).

3.6 Worst-case execution time measurement

Timing analysis relies on worst-case execution times (WCETs). These are obtained by the software component developer and are stated attribute of the component. The developer finds these in one of two ways:

- Static analysis using a model of the CPU.
- Measurement of the CPU running a test harness that drives the software down all the relevant pathways.

In both cases, a sophisticated CPU with non-deterministic execution time features will make the above much more difficult (in particular, obtaining an accurate CPU model becomes very difficult with complex CPU designs).

One of the other problems with finding CPU execution times is the behaviour of caches. The execution time in most CPUs with caches depends not only on the execution paths through the software but also on whether the software is interrupted and the caches disturbed: when the pre-emption from an interrupt is finished, the cache is in an unknown state and resumption of the software causes reloads. The extra execution time due to this is known as *cache-related pre-emption delay* (CRPD) and it is extremely difficult to predict.

As discussed earlier, worst-case performance is the key factor in an embedded real-time system and CRPD is major problem. This is why the cache described earlier in section 2.2 is partitioned: when each pre-emption occurs, there is a switch to a new partition of the cache. The CRPD is consequently zero and valid measurements of the execution time with the cache can be made and relied upon in the timing analysis. The cache is partitioned into four segments because a CPU core can typically be set up so that there are no more than four pre-emption levels (as described earlier in section 3.2).

WCET values must be enforced at run-time for the results of the timing analysis to hold. Because the WCET is not the elapsed time of an ISR or runnable it is difficult to handle in software. The new CPU core handles it directly in hardware.

An execution time counter (ETC) register counts clock cycles down to 0 and represents the remaining execution time budget. When an ISR interrupts, the ETC is stacked as part of the context of the interrupted ISR and a new value loaded representing the WCET of the new ISR. At the end of ISR execution the original ETC value is reloaded as part of restoring the context of the interrupted ISR and execution resumes.

Execution of runnables count towards the execution time of an ISR. But each runnable also has its own WCET and this must be enforced. This is accomplished by a modification to the ETC before and after each call to a runnable.

3.7 Executing runnables securely

A call to a runnable – from either the ISR or from another runnable – is a protection switch point: a runnable can misbehave but the effects of this must be isolated from other unrelated software components. A security stack is used to store enough of the current context to allow it to be reconstructed and execution of the caller to

resume. For example, on an Arm Cortex M device this includes the stack pointer and registers r4 – r11: these are the callee-saved registers defined by the Arm application binary interface (ABI). The C compiler trusts that the called function will ensure that these registers are unchanged over the function call. However, a secure environment cannot rely on trust.

Runnables are called through a runnable vector table (RVT). The vector table includes the entry pointer to the executable code, but it also includes other properties of the runnable:

- Protection registers (RVT, SVT and memory region protection)
- The boost priority (discussed earlier in section 3.2)
- The WCET of the runnable
- Worst-case main stack usage (including nested function calls but excluding nested calls to runnables)

The vector table is stored in normal memory (typically flash) and populated by the linker. A call to a runnable is performed by a CALL type instruction but instead of calling the entry function it is a call to the address of the vector entry. The call can be made by the ISR or by a runnable making a nested call. The CPU detects that this is not a regular function call and performs the switching operation.

The switching operation saves the current security context to a private security stack (not accessible from a runnable) and loads a new context. This includes the following:

- Saving the non-volatile CPU registers (i.e. callee-save registers, including the stack pointer).
- Saving current memory region protection registers and loading new ones. Memory region protection narrows accesses to regions of memory. The protection does not include address translation: the registers define the base and limit of a region. One of the regions defined includes the stack: access is restricted to a section of the current stack from its current top up to the maximum defined in the RVT.
- Saving base and limit registers pointing to the RVT and the SVT and loading new ones. Changing the RVT is used to prevent recursion but is also useful sandboxing feature that can be used to restrict access to certain device drivers. Similarly, changing the SVT restricts writes to signals published only by the runnable.
- Loading the ETC register. This is loaded with the lower of either its current value or the WCET of the runnable, and security stack context stores the difference between the old and new values of the ETC register.

A runnable is compiled as an ordinary C function and termination is via a normal RET type instruction. The CPU recognizes that the call is returning from a runnable function and performs the runnable termination operation.

Termination involves restoring the security context from the security stack:

- Loading the callee-save registers.
- Loading the memory protection and RVT registers.
- Adding the ETC difference to the ETC register.

Termination due to ETC being 0 is a fault that will result in the top-level runnable being terminated and ISR resuming.

The diagram below illustrates how the execution time is 'billed' to ISRs:

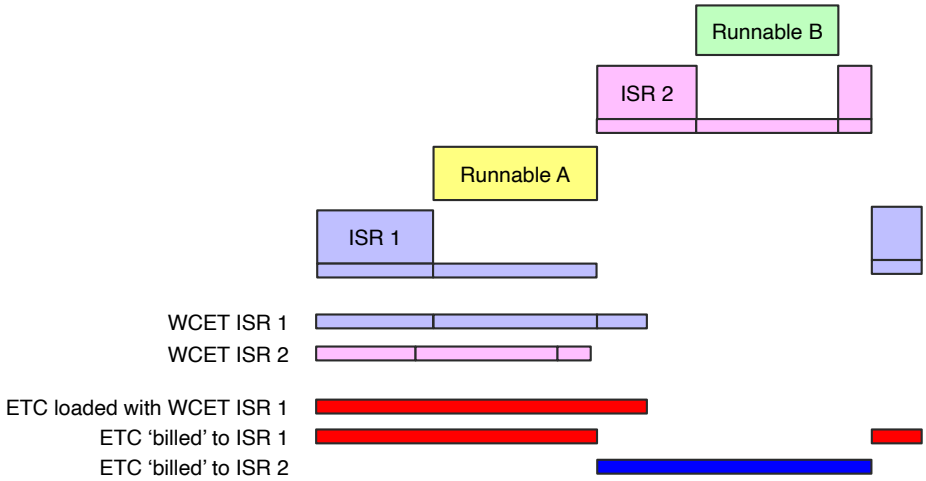

Figure 7: Recording execution time of ISRs

In the above diagram, the ETC register is loaded with the WCET of ISR 1 at the start. As execution proceeds, the ETC register is decremented. When a call is made to runnable A, the ETC register is temporarily reduced to the WCET of runnable A. When runnable A completes, the ETC is restored to what it would have been if A were just an ordinary function call.

Pre-emption by ISR 2 causes the current ETC register to be stored on the stack and loaded with the WCET of ISR 2. When ISR 2 finishes the ETC register is restored and execution continues and ETC resumes counting down.

4 Tool support

4.1 Compilation tool support

The AUTOSAR approach uses tools to generate C header files that are then used to compile the software component. The same approach is used here to generate code fragments that can access signals via meaningful 'handle' names and compile to code that accesses the SVT. Similarly, calling nested runnables can be done via handles that access the RVT.

Automatic code generation can be used at the integration stage to produce C source code defining storage for signals. This can be linked against each SVT, the data allocated to the dual-port RAM and the SVTs placed in flash memory.

The code for ISRs can also be automatically generated: once the mapping of runnables to an event (such as a periodic timer) is determined, the code to call these runnables can be generated.

4.2 Measurement tool support

The nesting of runnables can be determined from analysis of the source code (or from a specification of the software component) and the runnable vector tables generated for each runnable. A call graph of the runnables can be generated, which will bound the security stack depth, allowing it to be allocated automatically (the absence of recursive calls to runnables can be verified). Static analysis of the binary code for each runnable can determine its worst-case stack usage; this data can feed into calculations of the total stack depth of the complete firmware on a given CPU core.

Existing tools to measure WCET can be used. Two approaches are possible:

- The CPU is simple enough to be modelled in static analysis tools and a figure could be produced (using annotations for loop bounds).
- The execution times could be measured with a test harness to exercise all the pathways. The measurement process can use the ETC register for tracking execution times.

In the above approaches, the cache behaviour can be modelled or measured without needing to account for cache pre-emption delays: the cache can be assumed to be empty at the start of the execution of a runnable.

Response times can be calculated for all top level runnables using the execution model that is defined by how each ISR rate limiter is configured.

4.3 Optimization algorithms for allocation

The problem of allocating software components across many cores at first sight appears to be difficult. But it is straightforward:

- The raw data (WCET, runnable call graphs, deadlines for runnables, etc.) is known from the measurements of individual components.

- The allocation of runnables to cores and ISRs, and the base and boost priorities of ISRs are the key configuration parameters – all other parameters can be set automatically from this known configuration and the overall viability of a given configuration determined.
- Choosing the allocation and priorities is done via stochastic optimization techniques, such as simulated annealing or genetic algorithms. These techniques alter a configuration and use a *fitness function* (that assesses how well a configuration meets the requirements) to guide the optimization. The configuration is altered iteratively until a configuration is found where everything fits into the available resources.

This approach is straightforward: allocating tasks to a distributed system using response time analysis with simulated annealing has been demonstrated as far back as 1991 for integrated modular avionics (IMA).

5 Summary

5.1 Multi-core architecture

The new architecture is based on a very large number of simple CPU cores with the greatest DMIPS/MHz/mm^2 for the highest raw performance of a device. Software for the device is built from re-usable software components where the complete resource usage of a component can be known beforehand and the allocation of the components across the device performed beforehand by a tool using an optimization algorithm.

Each core is connected to four others and can communicate through dual-port RAM, with event signalling where writes can raise interrupts. Dual-port flash memory is provided in the same way, with caches on each side for performance. The caches are partitioned into four independent segments where the segment used is selected by the pre-emption execution level of the CPU. This enables deterministic execution of software in the CPU.

5.2 Secure signalling

A hardware mechanism for secure publish/subscribe signalling between software components is provided: setting the value of a given signal can be restricted to only authorized components. Similarly, reading the value of a signal can be restricted to only those components that need to use the signal. Signals are exchanged between cores via dual-port RAM. Signals needing wider distribution can be copied from core to core by software.

5.3 Protected real-time tasking

Software components are executed by a hardware real-time tasking implementation. The real-time requirements of each software component are guaranteed using timing analysis, and the assumptions of the timing analysis are enforced at run-time. Worst-case execution times are enforced and the rates at which interrupts are raised and software components invoked are capped by hardware rate limiters. This isolates a timing faults to the defective component.

As well as signal protections there are general memory protections that restrict the memory a software component can access. This includes stack protections so that all components can run on a single shared stack, and a defective component cannot corrupt the stack of another nor cause it to overflow its stack.

5.4 Configuration tools

For a given mapping of known software components to cores, the configuration and the worst-case timing behaviour can be determined at system build time: the timing analysis can be performed, and the allocation of dual-port memory calculated. This allows the viability of a specific mapping to be determined automatically by a tool. As a result, tools running optimization algorithms can be used to automatically fit a specified system into a device and for that device to be automatically configured to run all the software components securely.