TASKING®

ACHIEVING MAXIMUM SOFTWARE PERFORMANCE WITH AURIX AND AURIX 2G ARCHITECTURES

Dr. Alexander Herz,
Head of Product Engineering, TASKING

ACHIEVING MAXIMUM SOFTWARE PERFORMANCE WITH AURIX AND AURIX 2G ARCHITECTURES

Automotive software and function suppliers want to take advantage of the cutting-edge features of the AURIX™ and AURIX™ 2G architectures – but many are daunted by the challenge of porting their large codebases from their existing architecture to AURIX or AURIX 2G. Without help, reading tens of thousands of electronic control unit (ECU) manual pages and porting a large legacy codebase can seem nearly impossible. Other suppliers may be already using AURIX, but are struggling to fit additional functionality onto existing AURIX devices or to plan their transition to a different AURIX device. If these scenarios describe your situation, then you share a common problem with your peers:

"How can I fully exploit the possibilities of a new AURIX or AURIX 2G device so I do not fall behind the competition in terms of price, functionality, and performance, without sacrificing safety or investing man-decades to understand every single detail of the architecture?"

This paper explains how solutions and methodologies from TASKING can help you fully exploit the AURIX and AURIX 2G architectures. Some of these solutions and methodologies improve embedded software performance. However, performance is not the only relevant metric by which successful AURIX implementations can be measured. TASKING also provides tools that can help you accomplish the following:

- Ensure compatibility with popular microcontroller abstraction layer (MCAL) and real-time OS (RTOS) combinations that were created with older compiler versions
- Address the full range of AURIX and AURIX 2G devices while benefiting from the latest compiler technology
- Achieve superior performance on existing code without compromising safety
- Minimize cost and effort to correctly configure AURIX devices, including pins and the memory protection unit (MPU)
- Easily port code based on the popular Basic Linear Algebra Subprograms (BLAS), Linear Algebra PACKage (LAPACK), and Fast Fourier Transform (FFT) libraries to the AURIX platform (highly relevant to radar and sensor-fusion applications)
- Speed up your MATLAB-, Simulink-, or Model-based linear algebra computations
- Find concrete optimization steps in your hot code that boost performance up to several hundred percent
- Minimize the time your developers need to wait for debugger licenses during mass production

- Create a new memory layout or port an existing one
- Reduce cost and overhead of configuration management using position-independent modules that can be flashed anywhere

In the rest of this paper, we illuminate these tools and methodologies by discussing them in the context of the steps typically taken by automotive software and function suppliers when they port an application to a new architecture.

STEP 1: DRIVER AND RTOS SUPPORT

After choosing a specific AURIX or AURIX 2G device for a project, you need to select a compiler toolchain vendor and version that is compatible with the drivers, libraries, and RTOS versions for your target device. These typically include MCAL for AUTomotive Open System ARchitecture (AUTOSAR), SafeTLib and Secure Hardware Extension (SHE) for safety applications, and potentially additional libraríes for encryption, linear algebra manipulations, etc.

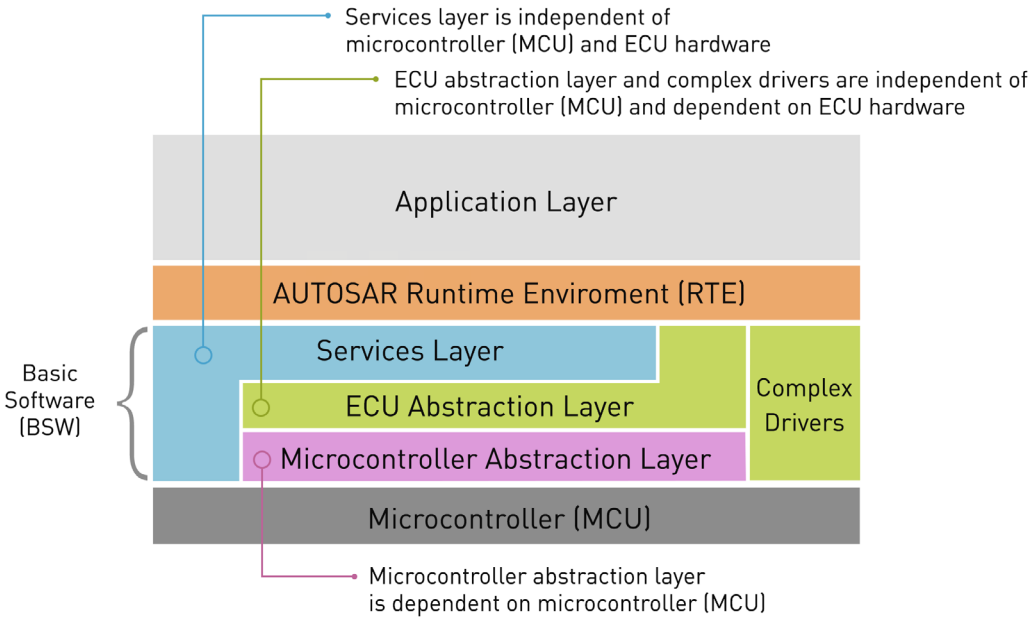

Figure 1

MICROCONTROLLER ABSTRACTION LAYER (MCAL)

As shown in Figure 1, the MCAL requires some careful thought about dependencies. In general, the target device vendor and RTOS vendor offer support for the TASKING compiler version of your choice. However, certification of all the necessary drivers and configuration files requires significant effort and time. In time-constrained situations, it may make sense to choose a combination of drivers and compiler version that is available off the shelf.

Some customers may have to use an older compiler version to compile MCAL drivers and other essential components, as these may have not yet been tested against the latest compiler version. For safety applications it is also important to note that the device headers (Special Function Register Addresses) to be used with a specific compiler version are tested and delivered as part of the MCAL drivers from Infineon. The device headers, which are delivered as part of the TASKING compiler release, should not be used when certifying a safety application. Safety is discussed in much more depth in the "Safety and Performance" section of this paper. If you want to use the MCAL or other pre-built drivers from Infineon, contact Infineon to determine which version of the TASKING compiler is appropriate for your device and your project.

You should consider using cross linking when using an older compiler release in order to utilize the MCAL and related libraries while benefiting from the latest compiler features and performance improvements.

CROSS LINKING

Cross linking enables you to safely mix binaries (such as MCAL drivers) created with older compiler versions with binaries of code compiled using TASKING v6.1r1 and later, as illustrated in Figure 2. Reusing old code and binaries can save time and money, while using the latest compiler optimizations for new applications can improve performance – allowing your application to fit into a smaller, cheaper device.

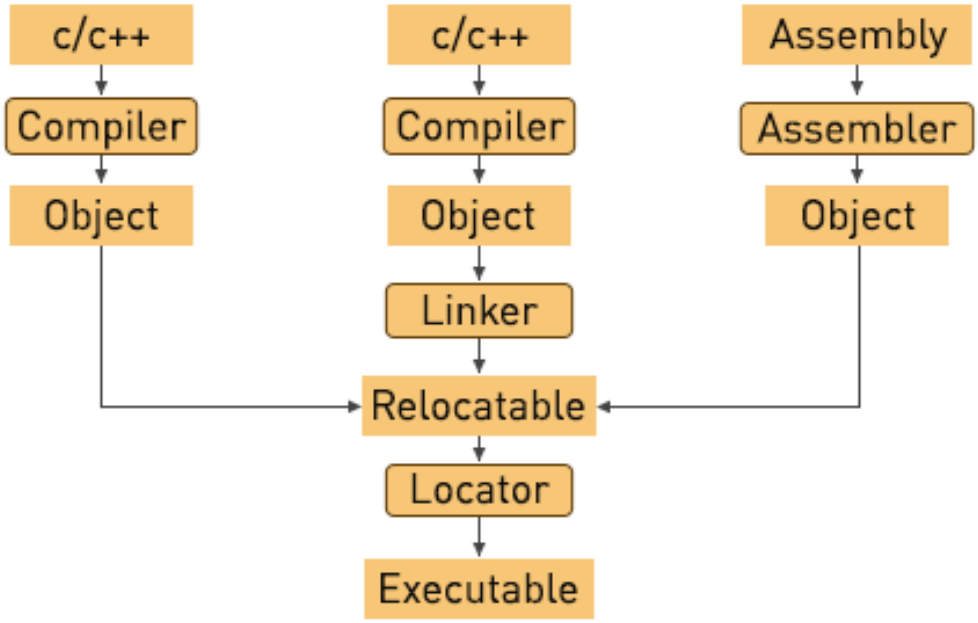

Figure 2

In addition to compiling the MCAL drivers with an older compiler release, any pre-certified older code can be reused unchanged and is compatible with TASKING's latest compiler release. Current applications and new code that is not bound to an old compiler version can instantly benefit from performance gains and improvements available with the latest compiler technology. Interoperability of old and new code is ensured by theoretical and thorough practical tests performed by TASKING between all relevant compiler versions, and by the user adhering to the cross-linking instructions in the manual.

STEP 2: SAFETY AND PERFORMANCE

Safety and performance of tools and their output are often handled as separate, discrete topics. Some developers may benchmark different compiler vendors to find the best tool for an application, then separately evaluate safety feature information. However, best practices mandate that safety should be built into a product from the ground up and is inseparable from performance. What is the value of having a tool with the best benchmarks on the market if you have to turn off many of the best features and optimizations because they may fail in corner cases?

SAFETY

ISO 26262-6 specifies guidelines for automotive product development at the software level (see Figure 3). To comply with these guidelines, you must choose software development tools that are safety certified. TASKING compiler and tool safety is ensured through the following means:

- Developed according to ASPICE CL2, guaranteeing the highest software quality through certified development processes and through quality assurance on all process and development steps

- Guaranteed, high-quality, low-latency support on current and old compiler versions so your development never stalls
- No third-party code in the C-compiler toolchain – all issues can be handled quickly in-house
- Available compiler options to help avoid triggering bugs that are present in the hardware
- Compiler toolchain safety kit that makes it easy to qualify and certify the compiler toolchain for your safety project
- TASKING Safety Checker tool that allows you to statically establish freedom from interference (FFI) between software components with different ASIL levels, according to Chapter 7 of ISO 26262

In addition, a combination of safety annotations from Safety Checker, together with our compiler technology, allows you to perform reverse inlining optimizations in a safety context. Otherwise, memory protection unit exceptions could be triggered if memory boundaries are disregarded.

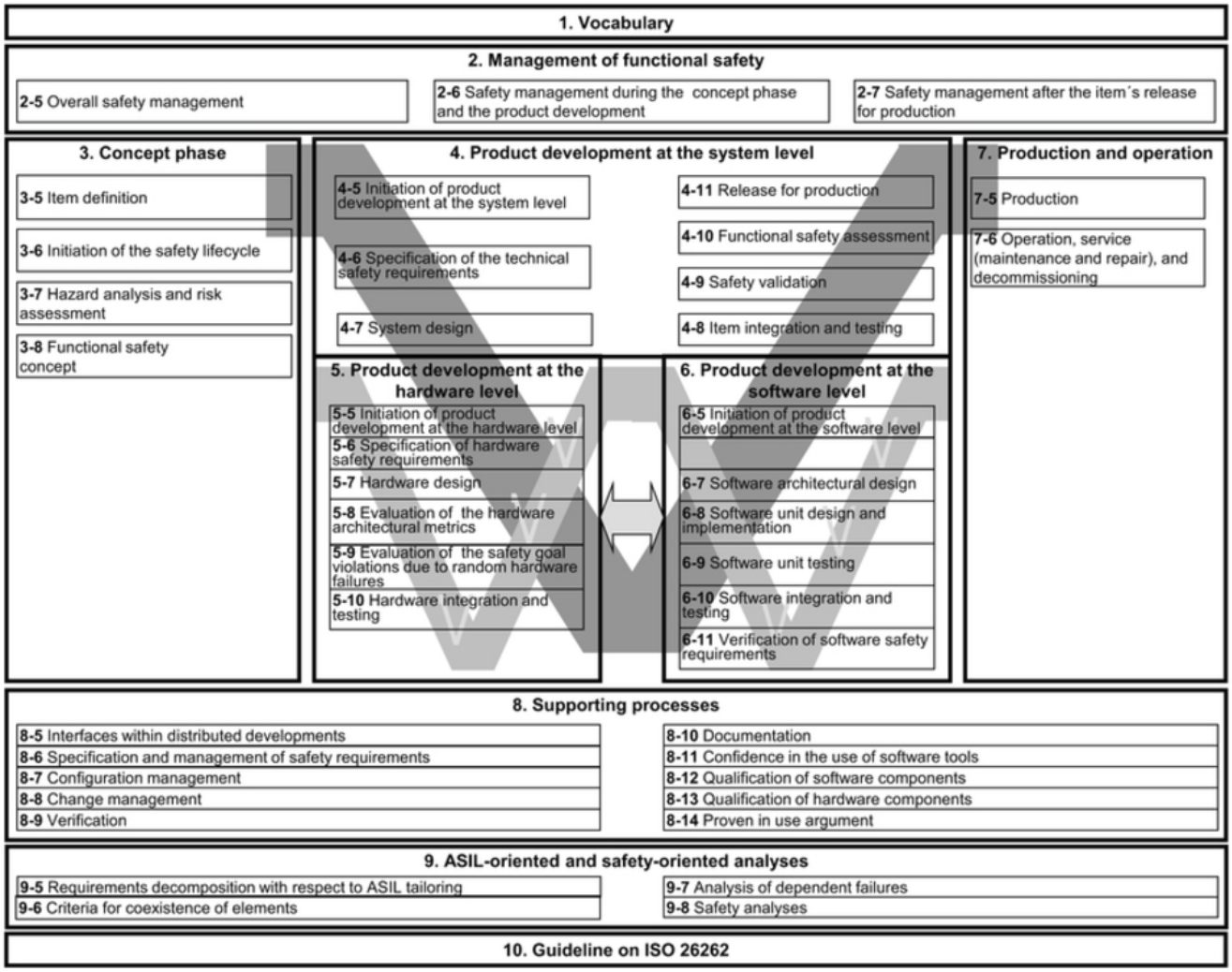

Figure 3

PERFORMANCE

You should look for more than a compiler that "just works." Instead, choose a compiler that benefits from its vendor's undivided attention. TASKING focuses specifically on developing the best compilers and related embedded solutions for the respective target platforms, rather than distributing the compiler as an add-on to an OS or an afterthought for some other product. Hundreds of man-years have gone into optimizing TASKING compilers and the underlying technology, thereby improving the compilers' performance.

Figure 4

COMPILER TECHNOLOGY

As shown in Figure 4, many tool options are available – making the correct choice depends on what you want to do.

Many embedded compilers are based on open source technology like gcc or LLVM. This can be an advantage when addressing high-end cores from Intel, Cortex-A, and similar vendors. From a compiler perspective, the performance-relevant characteristics of these architectures are fairly similar. Speed is typically determined by the following:

- Cache hierarchy (memory is much slower than the core and caches prevent the core from constantly waiting for memory) – see the discussion of the Intel Haswell and ARM v-8A architectures
- Efficient exploitation of the single instruction, multiple data (SIMD) units or floating-point units (FPUs)
- Other special-purpose instructions.

These high-end cores perform out-of-order execution, advanced branch prediction, register renaming, and many other tricks to improve throughput, which substantially lessens the pressure on the compiler in terms of pipeline optimizations, compared to smaller embedded processors. Many optimizations and algorithms in the frontend and backend of LLVM are tailored toward these kind of high-end cores. Porting code to a new core with similar characteristics can be as simple as adapting a few key values like cache sizes; the instruction set does not change.

Addressing complex embedded processors like the AURIX TriCore™ is dramatically different, because these processors are radically different from high-end cores. Often these embedded cores combine features from digital signal processing (DSP), reduced instruction set computing (RISC), very long instruction word (VLIW), and special accelerator hardware like the FFT unit in AURIX devices. Many algorithms from the existing LLVM backends are not easily adapted for complex embedded devices like the AURIX TriCore, meaning that the backend for such a device needs to be written almost from scratch. Because many of the performance-relevant optimizations for such complex embedded devices occur in the backend rather than the frontend, there is little benefit from using the LLVM framework to create high-performance code on an AURIX device. You are no better off than starting with a completely new compiler using a frontend of your choice. TASKING compilers are specifically created to effectively address all the quirks present in complex embedded processors. For example, there are many different ways to map C language expressions onto the various common and special hardware resources; the TASKING compiler can automatically choose the best option by balancing the user's size and speed preferences.

In addition, if a problem arises with an open source portion of an embedded solution, you must consider the availability of someone who can effectively provide the necessary open source support. TASKING avoids this issue by using technology that is developed solely in-house.

As a result, TASKING compilers are the top industry choice for AURIX and AURIX 2G architectures: they are selected for 80% to 90% of projects when benchmarked against the competition. One reason for their popularity is that they offer a 20% or greater performance gain over other compiler choices. As one customer stated, "... there was a significant runtime difference..." Another said, "... at the moment [TASKING results] look better than any other." Yet another customer added, "...we found that your compiler creates the quickest code."

As illustrated in Figure 5 a 20% difference can make the difference between fitting the application onto the target device or having to switch to a bigger, more expensive device.

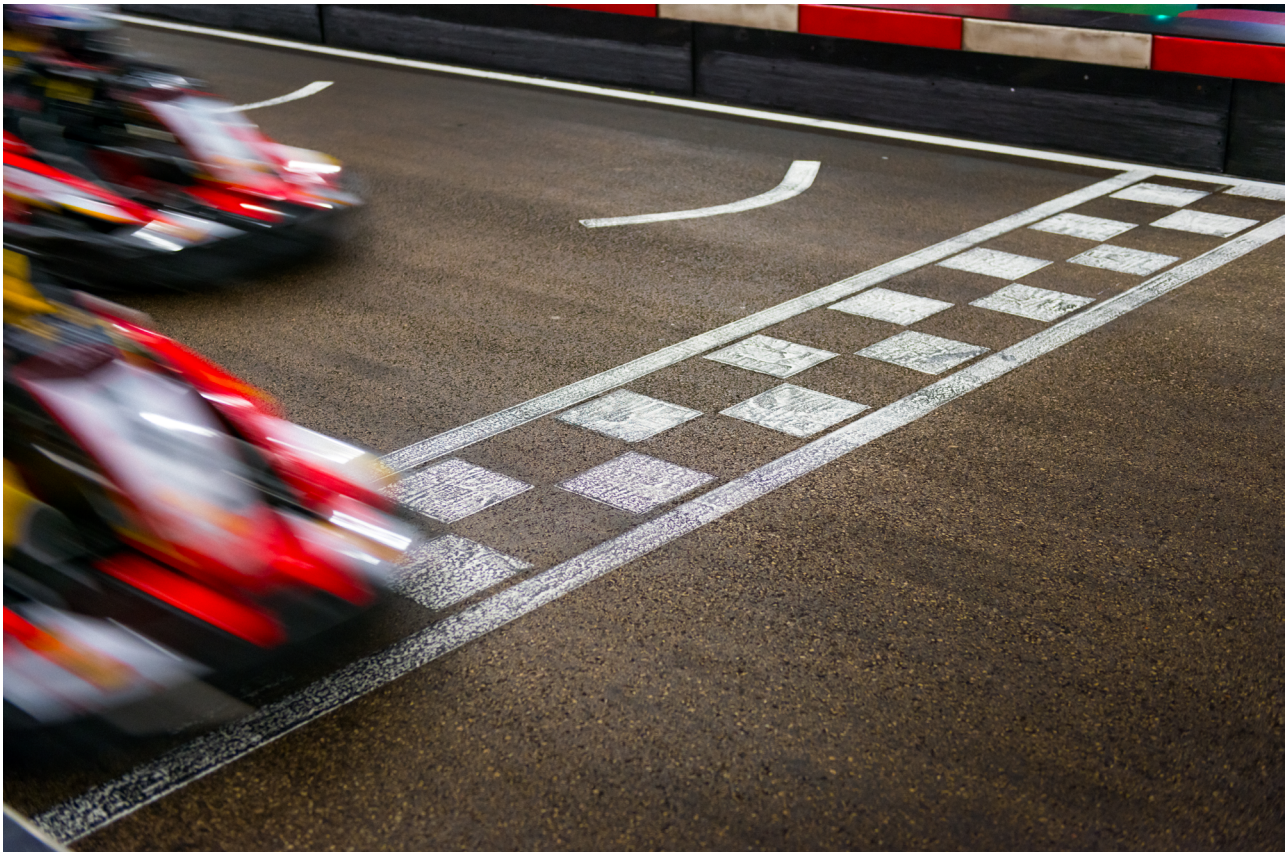

Figure 5

TASKING's industry dominance has attracted the support of numerous third-party and OS providers. These partners represent a continued industry investment, ensuring future compatibility between our products and a large sector of the automotive ecosystem.

You can explore how to perform meaningful benchmarks, using the discussion in the "Benchmarking" section. But before you can start benchmarking, you need to figure out how you can get your device up and running in a reasonable time frame without having to read several thousand pages of hardware manual.

STEP 3: DEVICE CONFIGURATION

The AURIX and AURX 2G are extremely powerful devices. There are hundreds of variants with different functional configurations. A single AURIX device can contain up to six TriCore cores, a hardware security module (HSM), a generic timer module (GTM), and a host of peripherals. Which all goes to show that the saying, "with great power comes great responsibility" can apply to more than socio-political discussions (or Spiderman).

A developer must write the startup code that correctly configures clocks, memory, and peripherals. An extensive manual is provided with each core, outlining pin constraints and peripheral configuration options – which all must be taken into account in order to create a valid pin mapping (see Figure 6). But especially in the case of benchmarking and prototyping, it is not viable to become an expert on all these details. Hence, TASKING created the AURIX Configuration Tools (ACT).

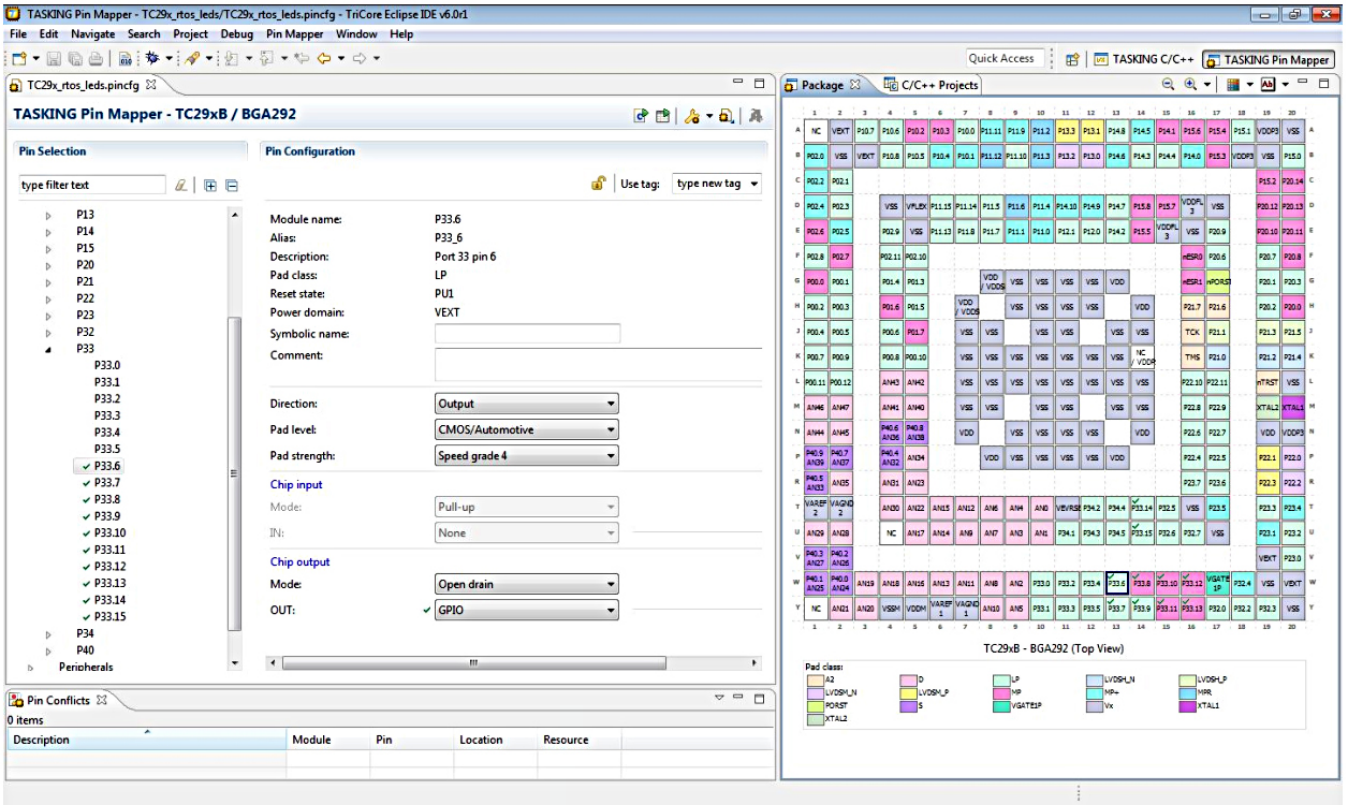

Figure 6

ACT consists of three components:

- A pin configurator that allows you to easily and graphically assign or swap pin functions
- A software stack that allows you to quickly and graphically configure peripherals like buses using the Infineon Low Level Drivers
- An RTOS

Using these components, a benchmark or a software architecture prototype can be configured and run within an hour, without the need to read thousands of pages in the manual. Furthermore, starting with v6.2r1, the safety annotations from the Safety Checker tool can be used to configure the MPU of the AURIX platform.

STEP 4: BENCHMARKING

Now that your device is correctly configured and running, you want to find the toolchain that gives you the smallest or fastest code for your application. Typically, users employ off-the-shelf benchmarks or an existing application with similar characteristics to the one they want to build. Then, they compare run times as well as disassembly of the components to determine the overall speed of their application as produced by different compilers. The goal is to discover which tool produces smaller or more efficient code and if are there any issues in the code generated by the compiler.

You should also consider how much effort you are going to have to spend moving your existing code to a new compiler. Porting guides – that simplify moving from your existing compiler to the TASKING compiler and result in clean measurement instead of the cable madness shown in Figure 7 – are available by contacting your TASKING representative.

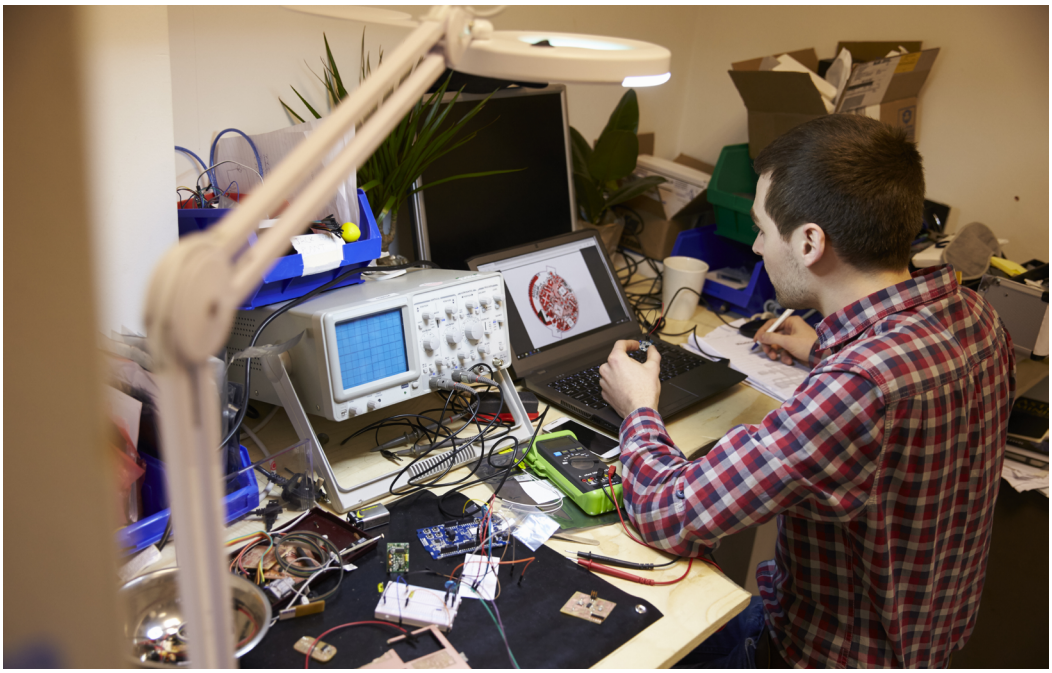

Figure 7

Here are a few common pitfalls you should avoid when benchmarking compiler toolchains so that you obtain relevant results:

- Standard benchmarks like EEMBC are heavily optimized by all compiler vendors, they do not necessarily give a good indication of how your code or application will perform. You should choose a benchmark or an application that is similar to what you need to build.

- Memory layout has a dramatic performance impact; the exact same code with a slightly different layout of variables or code can run much slower. Refer to the Memory Layout and Fine-Tune Your Application sections for more details about easily optimizing for this.
- Are you optimizing for speed or size? Make sure to provide appropriate switches to the tools.
- Device clock configuration and startup code generally have a significant performance impact. This code often comes as part of the toolchain and may configure the clocks differently; see the Fine-Tune Your Application section to determine how to validate your clock settings without the need for an oscilloscope.
- Measure the kernel of your application and exclude vendor-specific debugging functionalities such as printf and malloc.
- Measure clock or counter ticks from the same counter rather than time. Translation from ticks to time depends on the clock speed and may not be correctly computed.
- Perform a sanity check on your results. For example, a measurement that ran for 10 seconds on your stopwatch on a 300 MHz core should produce roughly 10*300*10^6 ticks if the counter is running at full core speed. Note that some counters only run at an integer fraction of the core speed.
- Calculate some relevant statistics: caches and other hardware features may have warm-up times, resulting in the first run of a piece of code being much slower than the following runs.
- The compiler is not the only factor that affects speed. Make sure that optimized libraries for AURIX and AURIX 2G and associated hardware accelerators are exploited for your computational needs. These can include encryption, hypervisor technology, and linear algebra for radar and advanced driver assistance systems (ADAS) (see the Performance Libraries section).

As we have explored, it is easy to measure a set of clock ticks on an application and compare that to some other clock ticks. It is even easier to do it in a way so that the results are completely meaningless. By following the guidelines above, you can make sure that you quickly get meaningful results that help you to make informed performance-related decisions such as deciding which tools to use and what software architecture works best.

STEP 5: SOFTWARE PLATFORM

Usually, customers build a software platform for a new target device before actual project work and application development begins. The software platform consists of the following:

- A pre-defined set of tools (specific compiler version, debugger, IDE, and so on)
- Software components, including the OS, basic software (BSW) modules, complex device driver (CDD), and the MCAL

The software components are optimized for use with a specific target device so that adaptation and optimization of the BSW, CDD, etc., as well as tool selection, does not need to be redone for every new project on the same target device. The goal here is, of course, to minimize the effort for the actual project so that existing application software can be adapted with least effort to the new device while ensuring correct timing and good performance.

Using a “checkbox” approach (see Figure 8) can help ensure everything is ready for development.

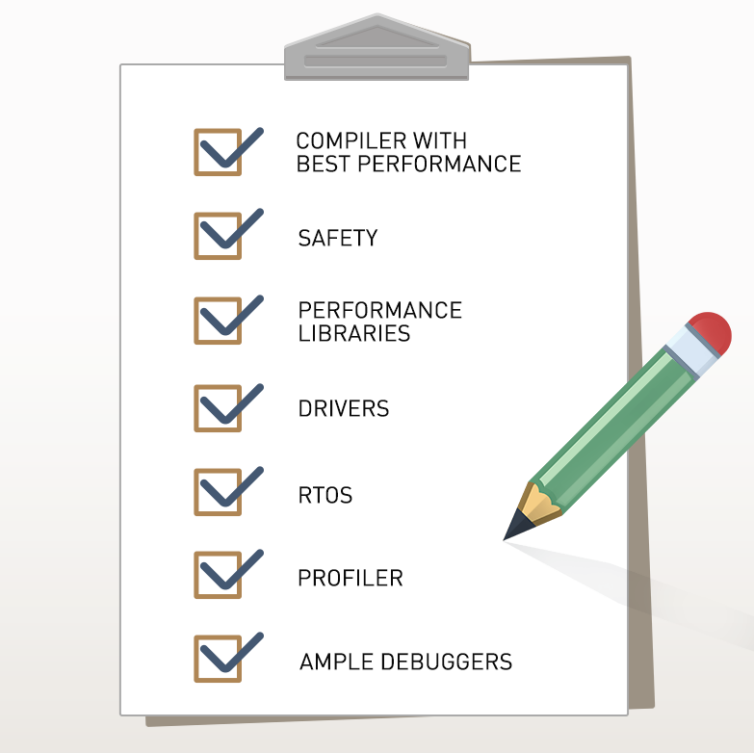

Figure 8

PERFORMANCE LIBRARIES

Before starting development, you must consider which functionalities you want to develop yourself and which are more effectively sourced from third parties.

Some functions are so critical to your application in terms of performance, numerical stability, or security that you want to employ deeply tested, expert-optimized and proven in-use libraries rather than spend man-decades implementing your own version from scratch. Often, these libraries are easily available on the desktop for prototyping on the C level or using model-based approaches. Unfortunately, support for desktop standard libraries is often missing when moving to an embedded device, or the available libraries are not suitable for use in automotive applications.

Figure 9

As shown in Figure 9, TASKING provides automotive-grade standard libraries and tools for embedded devices:

- Up-to-date, complete BLAS, LAPACK, and FFT (hardware and software, single-precision) for applications like radar, sensor fusion, MATLAB and Simulink code, and others
- Encryption algorithm libraries (HSM or software fallback with full AUTOSAR integration)
- Extremely low overhead (similar to a safety OS context switch), hard real-time hypervisor technology

By using the APIs of these standard libraries as building blocks for your application, you benefit in multiple ways. Porting the application becomes a breeze: the same application can be compiled on the desktop and for different targets. You do not need to port and optimize the underlying libraries yourself, and a wealth of information exists that explains how to use these standard libraries efficiently. In addition, you get high-quality support from experts in case you run into any problems.

A DEBUGGER WHEN YOU NEED IT

When selecting the tools for your software platform, there are many considerations to take into account. For safety projects, all tools that have an impact on safety must be qualified and may need to be certified. Good support in terms of safety kits, proven in-use technology, and proper development processes for the tools themselves are a must. Static analysis tools are highly recommended for both general code correctness and Freedom from Interference (FFI), depending on the ASIL levels of your software components.

Tools need to be cost-effective in they way they are actually used. If you have high resource demands or there is a risk that your resource demands may outgrow your target device, then you need the best compiler possible. When your applications easily fit into the device and you cannot use a smaller one, then safety measures may be of more relevance to you. These include safety kits; safety analysis functions like MISRA C, CERT C, and Safety Checker as well as reliable and proven in-use technology developed with certified processes.

The same is true for your debuggers and trace analysis tools. When highly trained expert engineers work on adapting the BSW, OS, and other key components to a new device, then only the best debugging tool with a wide range of expert features will suffice. The performance of all projects on this device depends on the quality of these components because the expert engineers are using these tools to their full potential.

Later, application software is usually adapted and extended as part of a specific project that is based on your software platform. In this stage, usually less hardware-savvy engineers, who are more knowledgeable about the function and the architecture of software components, are working on the code. Often, these engineers experience bottlenecks (see Figure 10) in their work because there are not enough licenses available for expensive, high-end debugging tools. Even if a license is available, the high-end tracing and timing debug features are usually overkill at this stage of development.

So instead of making your application developers wait for high-end tools, which they typically do not use to their full potential, you should consider cost-effective debugging solutions for the project phase. It is often possible to obtain six or more licenses of the TASKING Embedded Debugger, which are adequate for debugging in the project phase, for the same price as a single high-end debugger license.

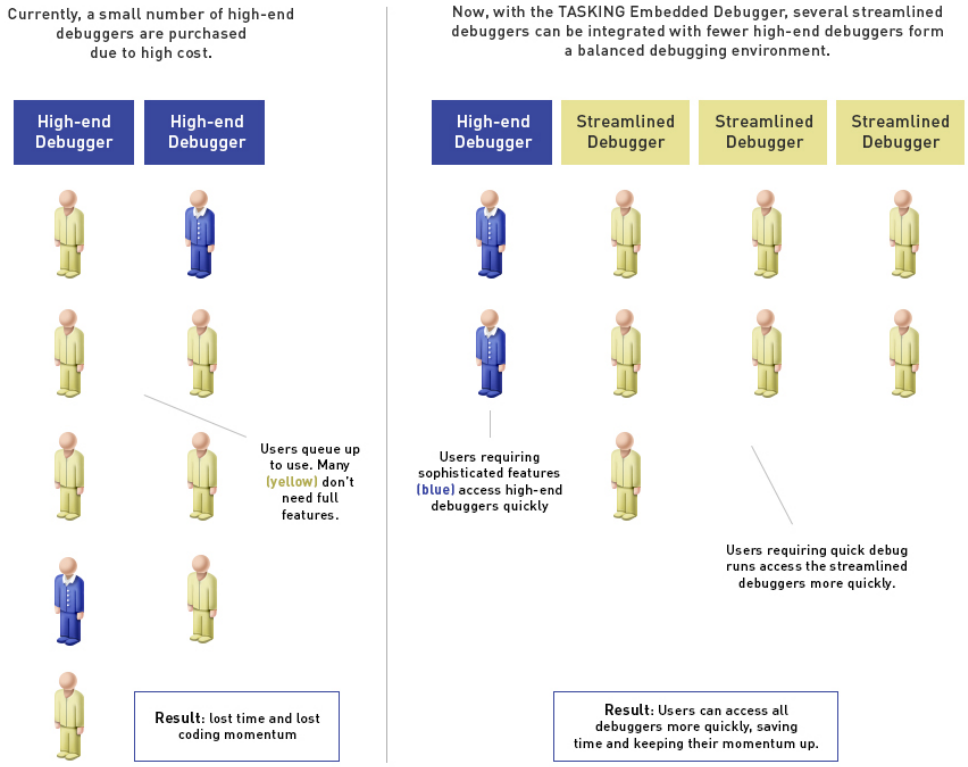

Figure 10

STEP 6: FINE-TUNE YOUR APPLICATION

After your application has been verified, thoroughly tested, and debugged, and by itself behaves correctly, you may still run into performance and timing issues when the application is integrated into the target environment. Many timing issues can be addressed simply by improving the performance of the runnables that cause a missed deadline. Furthermore, by reducing the core load of your applications, you may be able to select a less expensive device because it has fewer cores. An easy way to address these issues is performance tuning.

Performance tuning involves optimizing your application for a specific target device. Common situations where performance tuning of your application makes sense include:

- You are using self-made libraries that are called frequently and thus have a substantial impact on overall application performance.
- You develop or adapt low-level drivers and BSW or OS components.
- You are close to or above your core load budget limit.
- You have a timing problem in your schedule that could be fixed by speeding up specific tasks, and you want to avoid changing the schedule.
- You want to use a smaller ECU to save expense.
- You are committed to easily and cost-effectively tracking and improving the performance of your code on target devices.

On the desktop, performance tuning is a standard procedure applied to all performance-critical code, using tools like vTune. The new TASKING Embedded Profiler makes the same functionality available for AURIX and AURIX 2G devices. The tool provides an overview of the current clock settings at the press of a button – there is no need to use an oscilloscope to verify that the clocks are configured properly for a benchmark run. After verification of correct clock setup, you are guided through a few easy steps that pinpoint the source lines that cause the greatest slow down . The tool indicates the root cause of the slow down and gives simple instructions on how the problem could be solved.

After applying the suggested mitigation, the tool can be used to confirm that the problem has indeed been fixed. All this happens non-intrusively with real data collected from the application running on the real device. Using such a performance tuning tool, non-expert users can often speed up previously untuned applications by several hundred percent in just a half an hour.

STEP 7: MEMORY LAYOUT

Most projects are based on code from previous projects, as it is more efficient to reuse and extend functionality than to reinvent it. But even if you start from scratch, a software architecture is usually developed as proof of concept before starting the real project. When moving old project code or a new architecture concept to a new device, the application needs to be fit into the various device memories. Typical embedded target devices like the AURIX and AURIX 2G are very sensitive to memory layout; in extreme cases, misplacing a single datum can reduce the performance by a factor of roughly 10. Moreover, there are many different kinds of memories with different access speeds depending on which core is accessing which memory and whether additional cores access the same memory at the same time.

Hence, the system integrator that is responsible for the memory layout requires precise control over the memory layout as well as excellent knowledge about the memory architecture of the target device. The memory layout is commonly defined using linker scripts, called .lsl files (see Figure 11), which can reach sizes of 50,000 lines or more in large projects.

```
#if defined(__PROC_TC27XC__)
#define __REDEFINE_ON_CHIP_ITEMS
#include "tc27xc.lsl"
processor mpe
{
    derivative = my_tc27xc;
}
derivative my_tc27xc extends tc27xc
{
    memory dspr0 (tag="on-chip")
    {
        mau = 8;
        type = ram;
        size = 112k;
        map (dest=bus:tc0:fpi_bus, dest_offset=0xd0000000, size=112k, priority=1);
        map (dest=bus:sri, dest_offset=0x70000000, size=112k);
    }
    memory pspr0 (tag="on-chip")
    {
        mau = 8;
        type = ram;
        size = 24k;
        map (dest=bus:tc0:fpi_bus, dest_offset=0xc0000000, size=24k);
        map (dest=bus:sri, dest_offset=0x70100000, size=24k);
    }
    memory dspr1 (tag="on-chip")
    {
        mau = 8;
        type = ram;
        size = 120k;
        map (dest=bus:tc1:fpi_bus, dest_offset=0xd0000000, size=120k, priority=1);
        map (dest=bus:sri, dest_offset=0x60000000, size=120k);
    }
```

Figure 11

Manually determining a good memory layout usually takes man-months. First, an accurate model of the target memory architecture needs to be extracted from the several thousand pages of device manual. Then, the existing layout from the old project or the new concept needs to be fitted into the model (partially using placeholders for things which are yet to be developed) while taking into account as many performance, safety, and other constraints as possible. Finally, a large linker script needs to be created and the whole thing must be tested and iteratively adapted to changes in the project and optimized by moving many thousands lines of linker script.

TASKING tools support you in two ways to make memory layout a lot less painful:

- The profiling tool discussed in the previous section can easily pinpoint the root causes of the most common and highest-impact memory layout inefficiencies and suggest a simple mitigation so that you can optimize your layout quickly and efficiently.
- The TASKING graphical linker layout tool supports the necessary, relevant workflows required for a successful memory layout but abstracts the hassle of manually copying and pasting thousands of linker script lines.

TASKING experts can also advise on memory layout, working with your specific workflow requirements.

STEP 8: ONE SIZE FITS ALL – POSITION-INDEPENDENT MODULES

After having built a high-performance application using the methods discussed previously, you need to deploy your application. This can be difficult in the test lab, with just a few vehicles. But deployment can become a nightmare for vehicles in the field. Next to security concerns like tampering and the cost of getting physical access to the vehicle, the single biggest issue is configuration management.

There are many different car generations with many different software configurations available. Each configuration needs special treatment even if the same functionality is to be deployed, because the software modules are pinned to exact physical locations in memory. Therefore, small differences in memory layout between the different software configurations require relinking the software module specifically for each target configuration. Deploying a module with incorrect memory layout can compromise the safety of the whole car.

Figure 12

Over-the-air (OTA) updates using Global System for Mobile communication (GSM) links are becoming more common. This trend allows updating software without going to the dealer, making the whole process much more cost-effective and simple for the car owner. This will also lead to an increase of software update frequency and thus increase the number of software configurations that need to be handled, as not everyone will receive all updates, depending on availability of GSM and update plans.

Clearly, a method to reduce the burden of configuration management is needed. This is where the TASKING compiler support for position-independent modules will be indispensable to you. The TASKING compiler allows generation of software modules (code and data) that work correctly regardless of where in memory they are placed. As a result, the module can be combined with any software configuration, as long as enough free memory is available. This is the first implementation of position-independent modules for the TriCore architecture which does not require a runtime linker.

SUMMARY

As this paper has shown, it takes a great compiler – and a lot more – in order to benefit from all the potential offered by the AURIX and AURIX 2G architectures. By taking advantage of the available experience and solutions from TASKING, you can make your development tasks much easier. Of course, we are always looking for ways to improve, and tools are often better adapted to certain workflows than others. If you find yourself thinking "this would be perfect if …", then please get in touch and tell us!