



The 12 fundamentals of embedded software development

A case study on what you need to know
from first inspiration to masterclass

Contents

Improve time to market with confident quality	3
1. Code Size	5
2. Code Performance	8
3. DevOps	10
4. Debugging	13
5. Code Quality	15
6. Access to Support	19
7. Development Environment	21
8. Compliance & Safety	23
9. Licensing	26
10. Conclusion	28

We are the world leader of software and services for embedded system's development. We enable our customers to create and secure the products of today and the innovations of tomorrow.

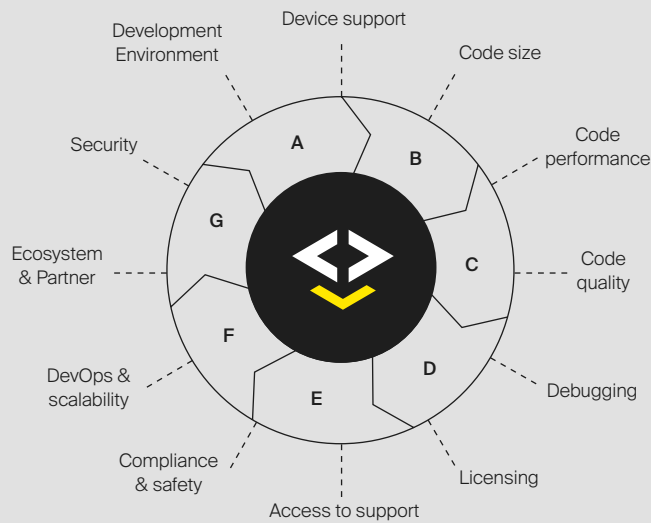
For more about IAR and our services visit iar.com

Improve time to market with confident quality

Is it possible to speed up time to market, secure quality, and at the same time stay within the budget? Particularly in the development of embedded software, which ensures product differentiation and thus a product’s success in the market, companies must weigh which investments lead to a clear ROI (Return on Investment) and a reasonable TCO (Total Cost of Ownership).

IAR Embedded Workbench

23+ architectures, one environment



- A Global technical support
- B Debugging & trace probes
- C Static code analysis
- D Runtime analysis
- E Safety Certification
- F IP protection
- G Production control

From consumer electronics products to automotive applications and industrial equipment: Customers constantly demand more and new features from products in ever-shorter cycles. These requirements from the market have a direct impact on the embedded software, which is instrumental in product differentiation, and its development. Embedded applications have become more complex than ever before and are built in large and distributed teams with different skills. There are many challenges and concerns to be addressed to meet the embedded application requirements, but still, developers need to be able to focus on innovation and make the best out of the product for differentiation in the market.

For 40 years, IAR has been a part of embedded software developers’ daily working routine and has a profound understanding of their processes. Not only does the fundamentals of embedded software development as IAR see them affect the productivity, efficiency, and quality of the product to be developed, but also the cost and time to market.

The 12 fundamentals of embedded software development

- 01 **Development Environment:** preferably an all-in-one IDE with project management tools and editor
- 02 **Device Support:** from many vendors including 8-,16-,32- and 64-bit cores connected to various projects in parallel and with different requirements
- 03 **Code Size:** by optimizing the application, companies could save money by using a smaller device
- 04 **Code Performance:** for faster code and a better user experience
- 05 **Code Quality:** translates into better products by following the best programming practices
- 06 **Debugging:** the key to enable full control of the application in real time to remove bugs and improve quality
- 07 **Licensing:** plus easy license management enable the customers to pay exactly for their needs from single users to license pools
- 08 **Access to Support:** essential to make sure programmers can focus on their code and get assistance and training when needed
- 09 **DevOps & Scalability:** addressing the growing demand and organizations need to modernize their infrastructure towards automated CI/CD workflows
- 10 **Compliance & Safety:** mandatory to prove that companies are compliant with specific requirements in their sectors
- 11 **Ecosystem & Partners:** benefits customers and provides the assurance that new devices, middleware, and integrations will be supported in future
- 12 **Security:** mandatory and companies are looking on how to implement security in the early and even late stages

The embedded development solutions from IAR cover all the embedded software development fundamentals, adding the value of increasing productivity and efficiency, securing the quality, and accelerating time to market.

This all comes at a cost that can be justified in the Return on Investment (ROI) and Total Cost of Ownership (TCO) use cases. In the following, specific cases show how taking the fundamentals into account has a positive impact on ROI and TCO. Security is not covered in the case studies as it deserves a full separate analysis. The topics “Ecosystem & Partners” and “Device Support” are covered in the “Development Environment” use case.

1. Code size

Why should you care about code size and benchmarks? By keeping your code size small, you can fit more functionality into a given device. By keeping track of your processor's benchmarks, you can use a cheaper device with a smaller flash size. So, both of these factors contribute to cost optimizations.

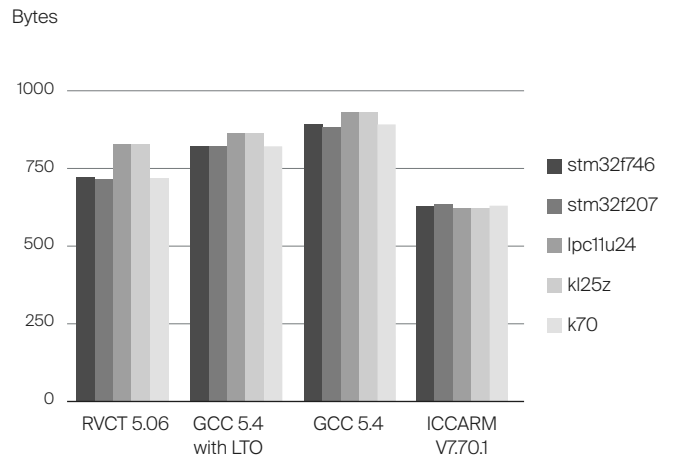


While [CoreMark](#) is a speed benchmark, it makes – thanks to its wide-ranging approach – a great benchmark for size as well. Looking at one file of the benchmark (coremark.c) on a variety of devices, we see a small degree of variability in the size depending on the device used displayed in the figure to the right. The tools, ARM RealView Compiler, GCC/GNU Tools ARM Embedded with and without LTO, IAR ANSI C/C++ Compiler for ARM are listed on the horizontal axis. The bars show the code size in bytes on different devices from NXP: Kinetis K70 (Cortex-M4F), KL25Z (Cortex-M0+) and LPC11U24 (Cortex-M0) and, from ST: STM32F207 (Cortex-M3) and STM32F746 (Cortex-M7).

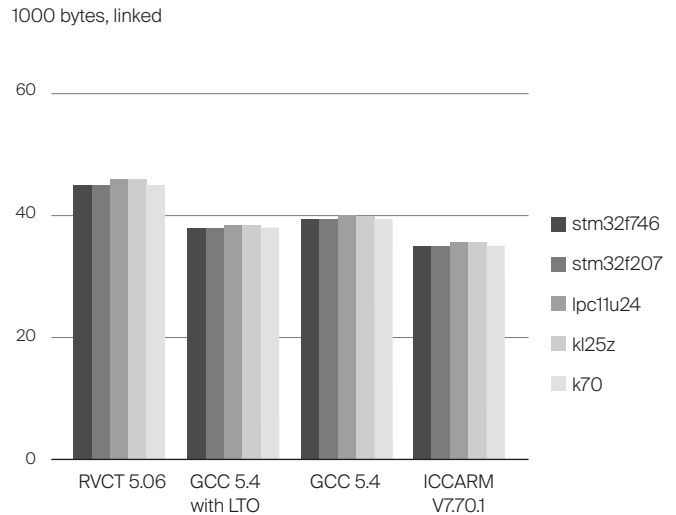
The IAR Embedded Workbench (shown in the bar chart: ICCARM V7.70.1) presents a much smaller degree of variability than the other tools, but a similar percentage size savings. The data is similar if we look at the matrix manipulation code (core_matrix.c) from the benchmark in next figure. Here we can see again, that the code compiled with the IAR’s tool suite is more compact than others.

In fact, on 30 out of 34 modules in CoreMark, the IAR Embedded Workbench for Arm produces tighter code, and the overall size difference is approximately 20%. Similarly, if we investigate the benchmarks for [IAR Embedded Workbench for RX](#) and [IAR Embedded Workbench for RL78](#) using real customer applications, we get 27% to 28% smaller code size than GCC and other tools.

Variability in the size by development tool and device



Variability in the size by toolchain



For continuous advice on how to improve developer efficiency, follow IAR Embedded Development on LinkedIn →



How much money can you save by going with a smaller part size? Obviously, this depends on many variables, including the underlying architecture and whether you can get a larger device with a similar peripheral set and package type. As an example, we will look at some popular Cortex-M4 devices from the same family and silicon vendor.

Considering the exact same MCU and peripherals, one device that has a 512kB flash sells for \$17.34 in single quantities from a major distributor (as of Nov. 2022) and a similar device from the same family with 1024kB flash sells for \$21.47 in single quantities from the same distributor, a difference of \$4.13 per part. If you can't

fit your code into the smaller device, you are going to necessarily spend 23.8% more on the silicon than you otherwise would have.

Even in a modest production run of 10k units, the added cost is \$41K. Doing various benchmarks from different silicon vendors like ST, NXP, Renesas, Microchip and TI, there is a price difference of at least \$1.00 when jumping from 256kB to 512kB or 1024kB for Arm cores or proprietary cores. Again, in a modest production run of 10K units, the added cost is at least \$10K. The price difference might vary a bit on the architecture and silicon vendor, but the total cost saving can be substantial.

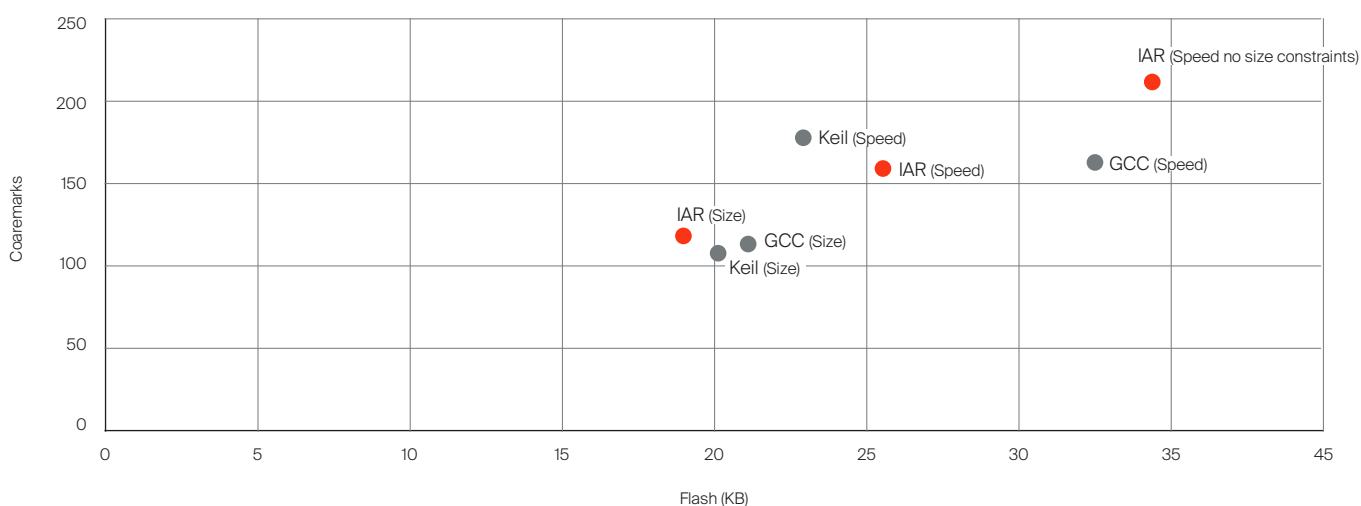
Discover how much code compression you can achieve by downloading IAR Embedded Workbench

Download →

2. Code Performance

How can the application performance influence your BOM (bill of materials)? How much of a performance bump can you expect from using the IAR Embedded Workbench vs. GCC- based tools?

Performance (Coremarks) vs. Code size (Flash)



Again, the [CoreMark](#) benchmark is a great reference because it tries to incorporate some of the more common things that developers do, such as matrix manipulations, CRC calculations, list processing (both find and sort), etc. As such, it gives you a “real world” comparison of what compilers can do, and it also has anti-tampering mechanisms to ensure that compiler vendors do not cheat by “hand-optimizing” CoreMark code. CoreMark benchmarks for a variety of MCU and compiler combinations can be found on EEMBC's website but let's take a look at some concrete benchmark performed by Nordic Semiconductor.

When compiled for pure performance, the IAR Embedded Workbench really separates itself from the

rest of the pack, particularly against GCC as to be seen above.

From these benchmarks, you can see that the IAR Embedded Workbench outperforms the Keil toolchain by 19.1% and the GCC toolchain by an astounding 29.8%. It is recommended to check the current and up-to-date scores at the CoreMark webpage. You can also run the benchmarks yourself to get the precise numbers.

But aside from the cost of a device, does this type of optimization really mean much to the average developer? To understand why you should care about this, let's perform a similar analysis to when we examined size optimization. Previously, we looked at

Performance Benchmarks

(Source: [Nordic Semiconductor](#))

Compiler	Coremarks	Coremarks/MHz	Code size (Flash, RAM) Bytes
IAR 7.30.4	212.0	3.31	Flash: 35449, RAM: 2273
ARMCC V5.17	178.0	2.78	Flash: 23600, RAM: 1672
GCC 5.2.1	163.47	2.55	Flash: 33336, RAM: 2824

two devices that were exactly the same except that one had a larger flash footprint to allow for more code if you were using a less efficient compiler. It's a little trickier to perform similar analysis based on maximum clock speed of the device since most parametric searches do not allow you to search at maximum clock speed.

However, we can compare similar Cortex-M4 devices from the same silicon vendor family, with the same packaging, same flash size and RAM size, number of 32-bit timers, number of D/A converters, etc. They can differ slightly in their communication interfaces (number of I2C and SPI for example), but the primary difference is their maximum clock speed (100MHz vs. 180MHz).

So, how much more is the 180MHz than the 100MHz? According to a major distributor, quite a bit. In quantities of 10k units, the price of the 180MHz device is \$3.78 per part and the 100MHz device is \$2.89 per part (as of Nov. 2022). This means a difference of 30.8% if you must go up to a larger clock speed to get the performance your application needs. For a production run of 10k units, that is a difference of over \$9K. As you can see, speed optimization can have an even larger impact on your BOM, particularly if you are producing in high quantities.

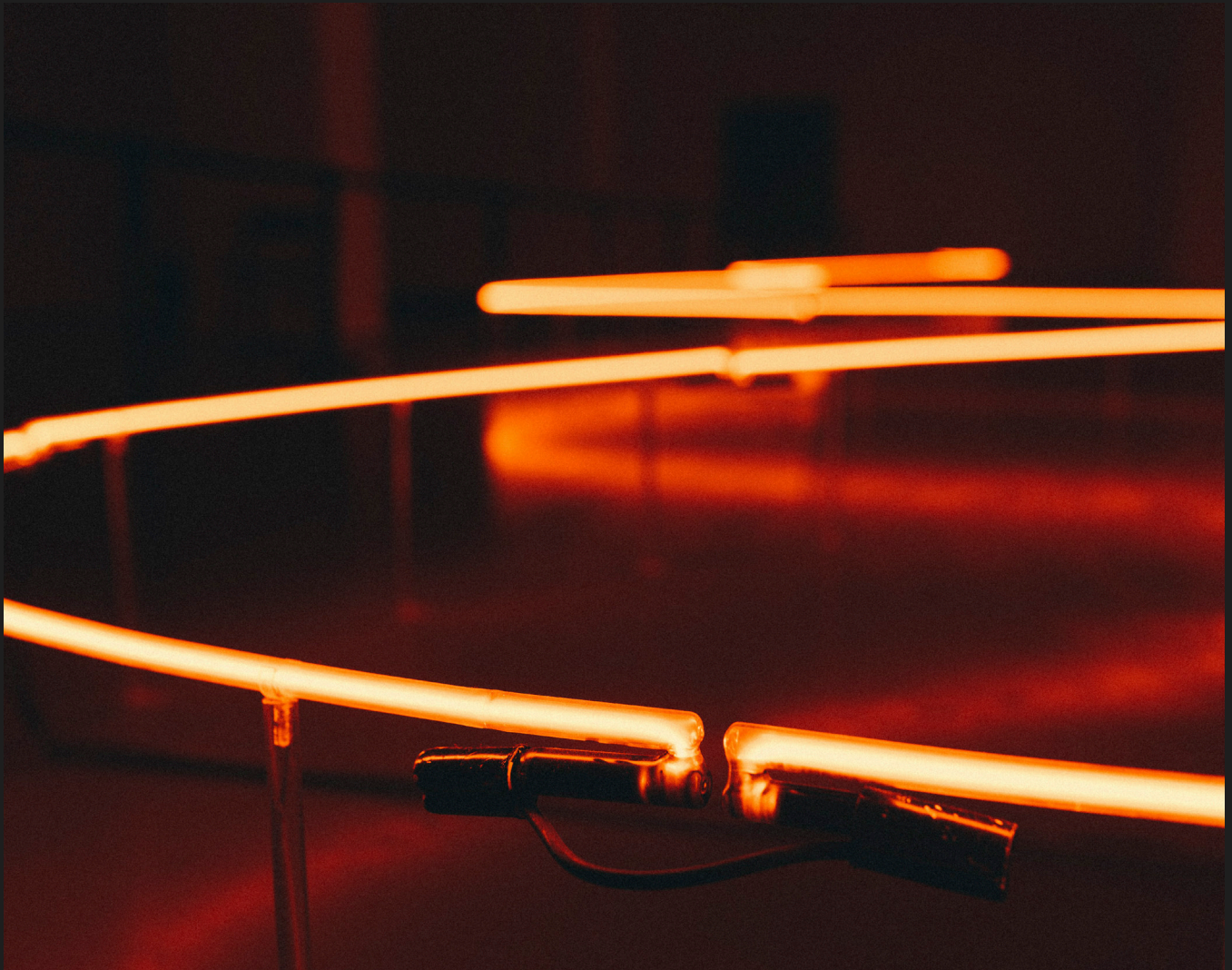


Measure how much performance improvement you can get by downloading IAR Embedded Workbench

Download →

3. DevOps

Lower compilation time to increase productivity. In general, each additional line of code or modification of software in the worst-case results in a full re-build of the software project in modern development workflows. In this case, if there is a huge code base, it takes a long time to build. As a result, the development time is increased by this waiting time.



How does this translate to your company?

Steve McConnell's book "[Software Estimation: Demystifying the Black Art](#)" contains a chart derived from the estimation model Cocomo II ([Constructive Cost Model](#)), which plots effort in man-months vs. size of the project in lines of code (SLOC). We can investigate the [COCOMO II Effort Equation](#):

Effort = 2.94 * EAF * (KSLOC)^E

EAF: is the Effort Adjustment Factor derived from the Cost Drivers.

E: is an exponent derived from the five Scale Drivers.

KSLOC: as measured in thousands of SLOC.

The EAF in the effort equation is simply the product of the effort multipliers corresponding to each of the cost drivers for your project.

Looking into the cost drivers extracted from the [COCOMO II - Model Definition Manual](#) in figure below, they have a significant weight. In the worst case, with extremely low rating levels the effect on the Effort Adjustment Factor (EAF) = 1.40 (1.20*1.17) to the best case when the EAF = 0.66 (0.84*0.78) with remarkably high rating levels.



Language and Tool Experience (LTEX) & Use of Software Tools (TOOL)

Source: [Rose-Hulman Institute of Technology](#)

LTEX Cost Driver

LTEX Descriptors	≤2 months	6 months	1 year	3 years	6 year	-
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Effort Multipliers	1.20	1.09	1.00	0.91	0.84	-

TOOL Cost Driver

TOOL	Edit, code, debug	Simple, frontend, backend CASE, little integration	Basic lifecycle tools, moderately integrated	Strong, mature lifecycle tools, moderately integrated	Strong, mature, proactive lifecycle tools, well integrated with processes, methods, reuse	-
Rating Levels	Very High	Very High	Very High	Very High	Very High	Extra High
Effort Multipliers	1.17	1.09	1.00	0.90	0.78	n/a

This will directly affect the productivity of your overall development team. The effect on your organization can be calculated and adapted for free at <http://software-cost.org/tools/COCOMO/>. The same applies for design and code generation tools: Longer build times for the automatic generated code impact the productivity on the design itself, as changes or new logic need to be tested and integrated into the overall system before proceeding with the design.

According to various customers feedback and also stated in the [customer story](#), the IAR Embedded Workbench running on Windows showed at least twice as fast build speeds compared to any other commercial tools. This is also valid for the IAR Functional Safety certified tools. The build times using IAR Build Tools for cross-platform support showed even better performance (4x times faster) when running on Ubuntu with the same hardware host. Performing the Standard C-STAT static analysis checks on Ubuntu took 25% of the time it took to perform on Windows.

Build and analysis results delivered faster means faster convergence to Continuous Deliveries (CD).

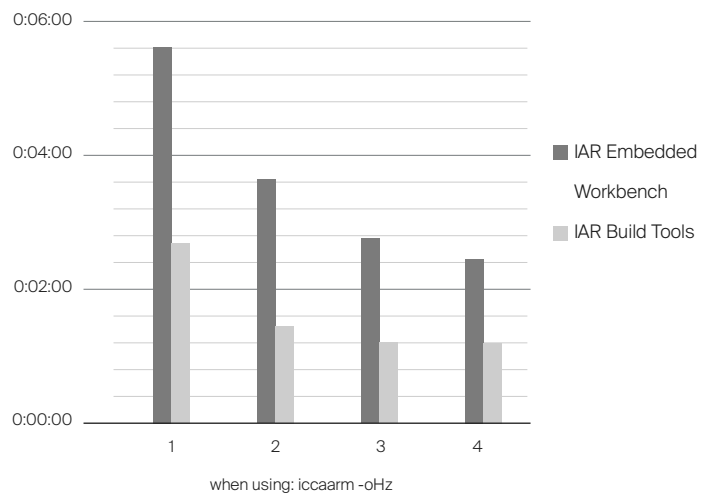
The build times displayed in the figure used:

- 574 C/C++ source files
- Highest compiler optimization level
- Analysis performed after project is built
- Comparison used the same host hardware, Intel i7-8700K, 24 GB RAM
- Using 1, 2, 4 and 8 CPU cores

Building embedded software projects with IAR Build Tools on Ubuntu is faster than building on IAR Embedded Workbench on Windows, generally taking less than 50% of the time to build the project.

Build Times for IAR Embedded Workbench vs. IAR Build Tools

Source: IAR



Additionally, there is an essential need for automated processes to ensure quality and run builds and tests continuously in modern embedded development workflows. Embedded software R&D teams can achieve shorter time to market for new features when proper DevOps practices are implemented with the same functionality from the underlying command line tools in cross-platform frameworks.

The IAR solutions support modern and scalable build server topologies on Ubuntu, Red Hat and Windows for CI/CD pipelines including Virtual Machines, Containers (Docker) and Self-hosted Runners.

Check how to increase your productivity by downloading IAR Embedded Workbench

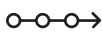
Download →

4. Debugging

To reduce debugging time, developers need to master advanced debugging strategies available on modern microcontrollers and supported by professional development tools. Here is your smart and advanced debugging features:



Integrated debugger
for source and
disassembly



Timeline
window



RTOS
awareness



Edit source files
without leaving the
debug session



Power
vizationalization



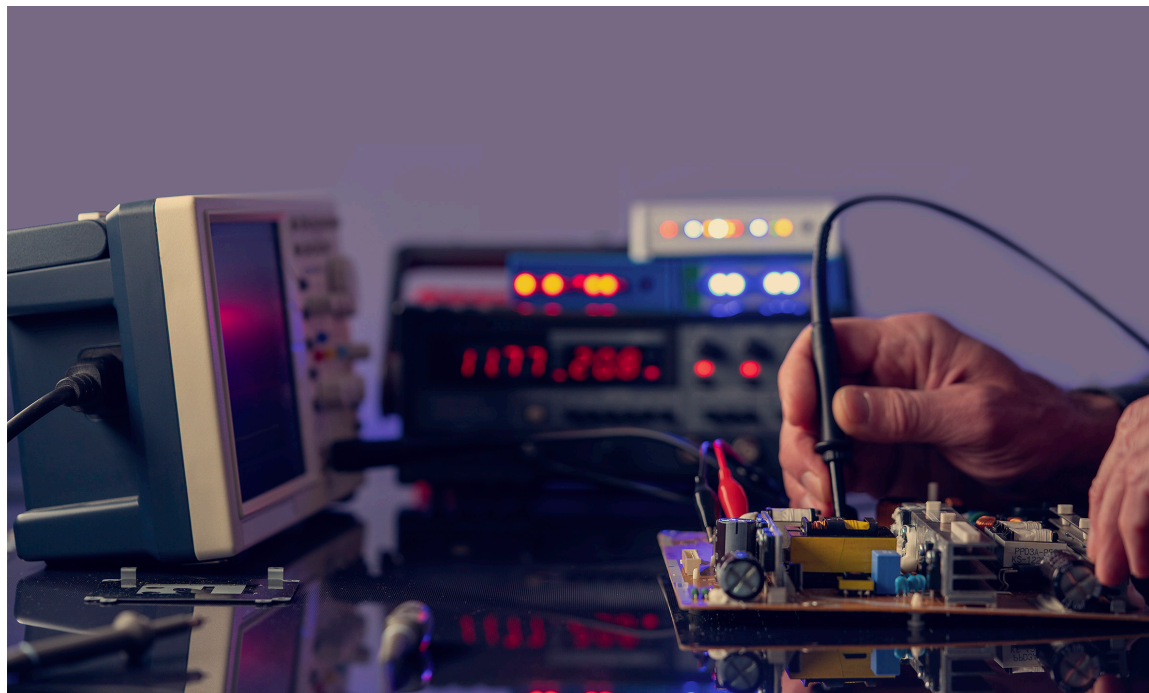
Dockable windows
and tab groups



Performance
analysis

The study “[An Exploratory Study of Debugging Episodes](#)” by Abdulaziz Alaboudi and Thomas D. LaToza observed that developers spend about half of their programming time debugging. Discussions at [StackOverflow](#) and [Reddit](#) claim even higher numbers of up to 80% or even 90% of a developer’s time spent on debugging. This sums up to about 1,000 to 1,600 hours a year for only one developer!

So if you think your developers spend their time on great innovations, think again: Most of your budget is spent on debugging, and if debugging takes too long, then releases and new versions are delayed. R&D is spending ever-increasing time on finding and solving problems on software systems, which are getting more complex by the minute.





Many programmers turn to general-purposes debuggers, such as GDB. These let the programmer step forward, inch by inch, through their code and set watchpoints as they go – which is probably the least efficient debug method known to humankind. Unfortunately, embedded software developers default to debugging with breakpoints and single-stepping, often due to tool limitations. To reduce debugging time, developers need to master advanced debugging strategies available on modern microcontrollers and supported by professional development tools.

The quality of a product will only be as good as the debugging capabilities that a developer has available. Can your team make sure to analyze and track the exact root of a specific bug? Are they fixing the issues or mainly applying workarounds using their best guess because the tool cannot provide enough detailed and real-time information? The state-of-the-art C-SPY

Debugger included with IAR Embedded Workbench gives full control of the application in real time.

Furthermore, it offers many smart features like complex breakpoints (code and data – unlimited), watchpoints, profiling, code coverage, timeline with interrupts, power logging, and even trace. Bugs can easily be exterminated from their source reducing the time spent debugging.

Mastering all these techniques and knowing when to use them can dramatically decrease how much time is spent debugging when a defect does get into the system. IAR has heard of cases where developers from partners have gone from debugging 80% of the time down to less than 5%. Taking a conservative approach in reducing at least two thirds of debugging time would mean max. 500 hours per year, freeing up a lot of person-hours (~1,000 hours) that can be reallocated increasing available developer time.

Try advanced debugging by downloading
the IAR Embedded Workbench

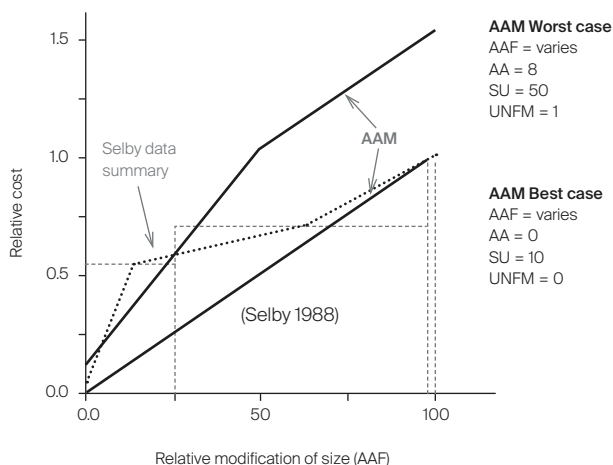
Download →

5. Code Quality

The cost of defects. On average – according to Steve McConnell’s book “[Code Complete](#)” – a developer creates 70 bugs per 1,000 lines of code. Roughly 20% of those – 15 bugs per 1,000 lines of code – will find their way to the customers. And the bitter truth is that fixing a bug takes 30 times longer than writing a line of code.

Boehm’s COCOMO non-linear reuse effects method

Source: [Rose-Hulman Institute of Technology](#)



By introducing code quality control early in the development cycle, the impact of errors and the effort for their elimination can be minimized. Providing static analysis right at the computer of each developer with well-defined coding standards can help them find issues in the source code during development, where the cost of errors is smaller than in the released product.

Additionally, many people talk about designing their code for reuse, but software estimation models claim reused code at being at least 50 percent of the effort of simply writing new code.

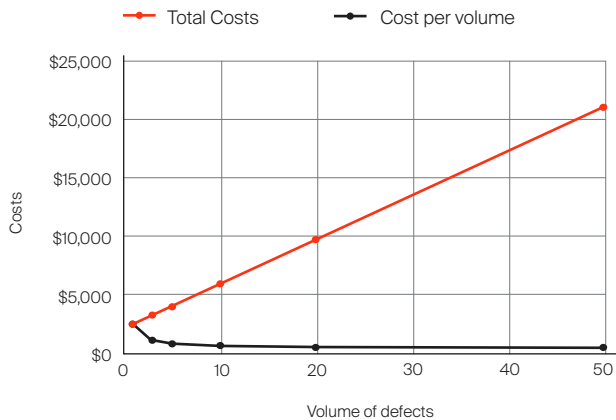
The [Boehm’s COCOMO](#) method shown on the left side estimates how the relative cost of writing the code is dramatically impacted by how much modification you do to the reused software in the dotted line. The x-axis is the percentage of modification you do to the code you intend to reuse while the y-axis represents the percentage of what it would be if you wrote fresh code. Note that for two of the three data samples of code, you did not have to modify much of the supposedly reused code to suddenly jump to 50% of the effort of rewriting the code from scratch. The AAM (Adaptation Adjustment Modifier) lines shows that small modifications in the reused product can generate disproportionately large costs. The key point here is that if you really want to reuse code, it has to be of remarkably high quality and well-designed in order to be cost-effective.



For continuous advice on how to improve developer efficiency, follow IAR Embedded Development on LinkedIn →

The total cost and cost per defect

Source: [Capers Jones: "Estimating Software Costs"](#)

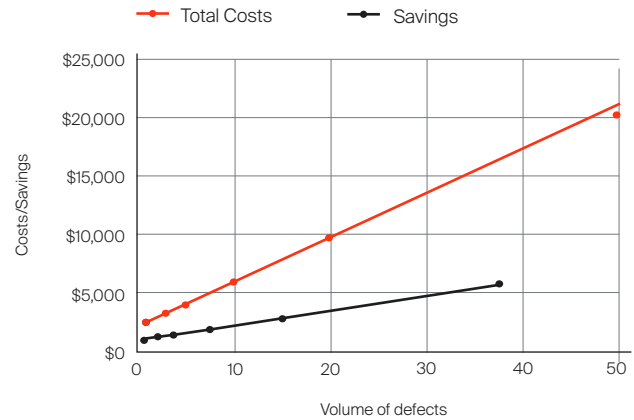


The fastest way to improve code quality is to use code analysis tools. In fact, if you are creating a functional-safety certified application, you are even required to use static analysis. These types of tools help you find the most common sources of defects in your code, but they also help you find problems that developers tend not to think or worry about when they are trying to write their code, especially when they are just putting up scaffold code to just get something working. These types of tools really help you develop better code because they enforce coding standards. Depending on the quality of your static analysis solution, they can check for many other potential issues while you are still writing on your code.

There are several reasons why code quality is a big issue. First, depending on the maturity of your development organization, developers can spend up to 90% of their time on debugging. If you could quickly isolate defects before they make it into a formal build,

The savings versus total cost of reducing the number of defects entering testing at each phase by 25 %

Source: [Capers Jones: "Estimating Software Costs"](#)



you would have a lower defect injection rate which means you can meet your organization's quality metrics much faster. Second, it also means that your code has fewer remaining bugs overall, which makes it a suitable candidate for reuse since using the code again has a lower chance of uncovering a previously undetected bug. High-quality code is easier to maintain because of fewer defects and – if it follows good software engineering principles – it will be easier to extend, therefore reusing it really does give you faster follow-on projects.

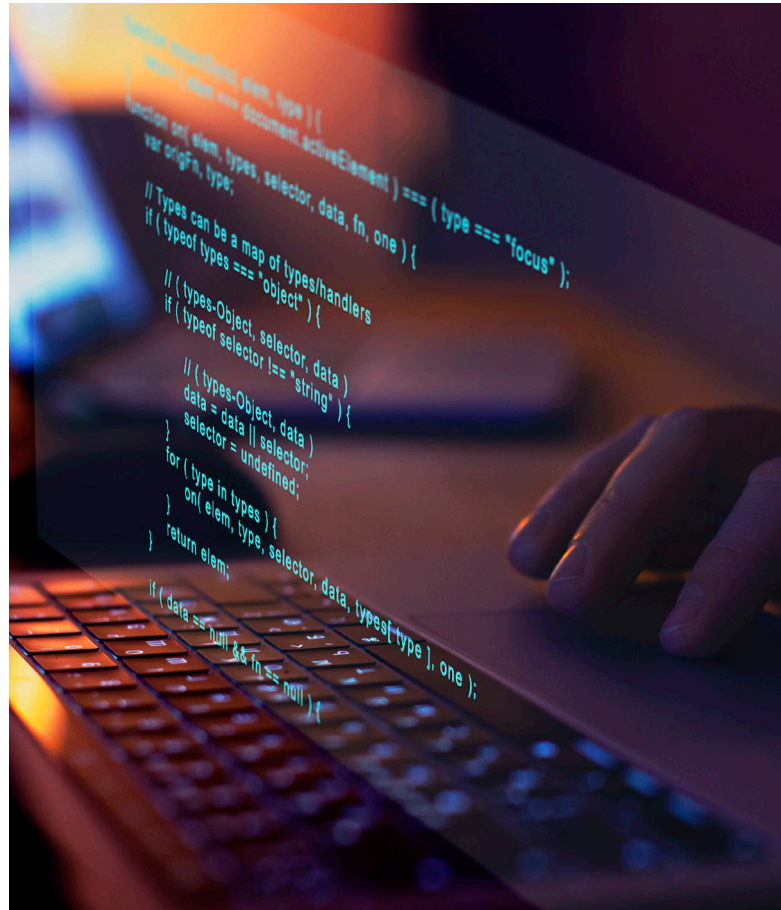
Why quality matters

What is interesting is that the cost per defect at each phase goes up as expected, but total costs are going down due to the decreased volume of defects in the figures above from Capers Jones' book on "[Estimating Software Costs](#)". In practice, it does not take longer to find and fix bugs at each phase, but the costs are still there despite diminished volume. It is worth noting,

also, that as a product matures into operation, the maintenance cost per defect is much higher due to the impact of servicing a fielded product. Other intangible costs such as damage to brand and loss of future customers and income, are still factors to consider.

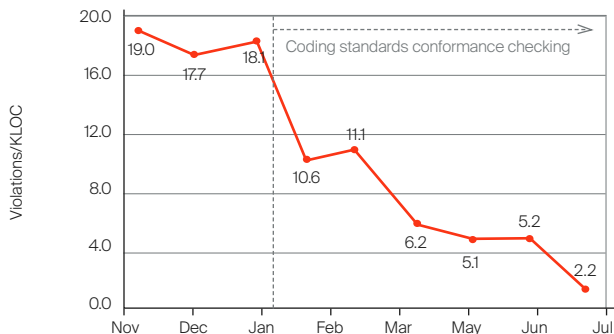
So, what is the return on investment considering these factors? Static analysis reduces the number of errors in software development at all stages of development. A simple analysis is to reduce the number of errors using the data in the figure from the previous page. Given this reduction in errors introduced during development, we see a significant cost reduction.

This simple analysis yields savings of about \$126 per bug, which – assuming an average of 15 bugs per 1,000 lines of code during development – translates to savings of \$1,900 per 1,000 lines of code. Of course, results will vary, also based on other factors such as labor rates, defect detection and repair time, and defect density. But since many systems use 10 to 100KLOC or more, the business case for static analysis is clear.’



Violations/KLOC

Source: [Dr. Dobbs](#)



Enhancing coding skills

Additionally, on another study done by Dr. Dobbs (figure to the left) pegs it as lowering defect injection by 41% – a massive savings in test time which goes straight to the bottom line not only in engineering time saved, but also accelerated time to market.

The injection rate in this study was quite constant from month-to-month until the organization introduced coding standards, then the defect rate dropped like a rock. As the developers became more familiar with the standard and had fewer deviations, the defect rate plummeted.



Google [published an article](#) in an ACM publication looking at the merits of code analysis. While the article takes a holistic view of their entire codebase including C, C++, and Java, the results are very clear:

“Compiler errors are displayed early in the development process and integrated into the developer workflow. We have found expanding the set of compiler checks to be effective for improving code quality at Google.”

The authors stated that moving static analysis checks into the compiler workflow and making them appear as errors, drastically improved the attention paid to the tool’s findings and that it ultimately meant that their code was of a much higher quality.

Furthermore, they discuss a survey sent to developers who recently encountered a compile time error and developers who had received a patch with a fix for the same problem:

“Google developers who perceive those issues flagged at compile time (as opposed to patches for checked-in code) catch more important bugs; for example, survey participants deemed 74% of the issues flagged at compile time as “real problems,” compared to 21% of those found in checked-in code.”

The article also points out the importance of having code analysis as part of the workflow by stating that when they automatically ran commits through a static analysis tool and invited engineers to look at the analysis dashboard, very few engineers followed through. Having instant feedback in the compilation process made static analysis easier to use and harder to ignore. Therefore, Google chose to integrate static analysis by default in everyone’s workflow. They believe that for code analysis tools to succeed, developers must feel they benefit from their use and enjoy using the tools. The point is that coding standards really do have an impact in development efforts.

Learn how to improve code quality and code reusability
by downloading IAR Embedded Workbench

Download →

6. Access to Support

Access to technical support pays off your development tools. What really distinguishes the quality of a professional tool is the quality of the technical support provided with local teams all over the world, speaking the customers' language.

If there is a problem with free tools, such as a bug in the compiler or in the libraries, the only thing that users can do is try to fix it by themselves or post an issue on the relevant repository. They hope that the problem will be fixed by the GNU community, or pay someone to fix or add the features. What this will really cost the user (in time and/or money) is impossible to predict.

Not having the entire development team stopped due to issues on the development tools is one of the biggest advantages of making use of the professional development tools from vendors like IAR. IAR provides easily accessible technical support with local support teams all over the world, covering several different languages such as English, Swedish, German, Korean, Japanese, Chinese, and Arabic. Customers get specified lead times and temporary workarounds to enable them to continue focusing on their application.

IAR will use reasonable efforts to resolve errors or reduce the severity level of the error via a workaround or a correction of the error in accordance with the repair time. IAR recognizes that errors defined as critical or serious can impose a major inconvenience for the Licensee, and IAR will therefore use its reasonable best efforts to provide a correction of the error as soon as possible, irrespectively of the repair times defined herein.

The figure in the next page shows repair times for regular Support and Update Agreement (SUA) customers, with a response time from IAR within one or two days and a repair time for critical issues in no more than 15 working days.



How do I get a bug fixed or a feature added?

There are lots of ways to get something fixed. The list below may be incomplete, but it covers many of the common cases. These are listed roughly in order of decreasing difficulty for the average GCC user, meaning someone who is not skilled in the internals of GCC, and where difficulty is measured in terms of the time required to fix the bug. No alternative is better than any other; each has its benefits and disadvantages.

- Fix it yourself. This alternative will probably bring results, if you work hard enough, but will probably take a lot of time, and, depending on the quality of your work and the perceived benefits of your changes, your code may or may not ever make it into an official release of GCC.
- Report the problem to the GCC bug tracking system and hope that someone will be kind enough to fix it for you. While this is certainly possible, and often happens, there is no guarantee that it will. You should not expect the same response from this method that you would see from a commercial support organization since the people who read GCC bug reports, if they choose to help you, will be volunteering their time.
- Hire someone to fix it for you. There are various companies and individuals providing support for GCC. This alternative costs money, but is relatively likely to get results.

Source: <https://gcc.gnu.org/faq.html#support>

Response and repair times within regular SUA

Source: IAR

Security level	Response time	Repair time
Critical	1 working day	No more than 15 working days
Serious	1 working day	No more than 30 working days
Moderate	2 working days	At next scheduled service or feature release, but not later than one year
Minor	2 working days	At IAR's discretion



The cost savings that result from an all-inclusive support and services agreement can be easily calculated, for example according to EMBECOSM's case study "[How much does a compiler cost](#)" based on GCC/LLVM: Toolchain maintenance requires a typical effort of 0.25 engineer months per month. There is a rule of thumb that the cost to employ is typically 1.25 to 1.4 times the salary depending on the benefits, payroll taxes and corporate liability insurance. Considering that, a compiler [engineer costs](#) on average \$117K per year in the US x 1.4, that would be \$3.4K per month for maintenance or per serious bug to be fixed.

The IAR Embedded Workbench license is provided with included Support and Update Agreement for 12 months. Subsequently, it costs customers only 50% of the own monthly maintenance cost per year to keep the SUA active. Not considering the fact that the development team stays productive at all times since the IAR team will be able to provide workarounds in one to two working days.

Explore IAR's embedded expertise by checking the resources and documentation by downloading IAR Embedded Workbench

Download →

7. Development Environment

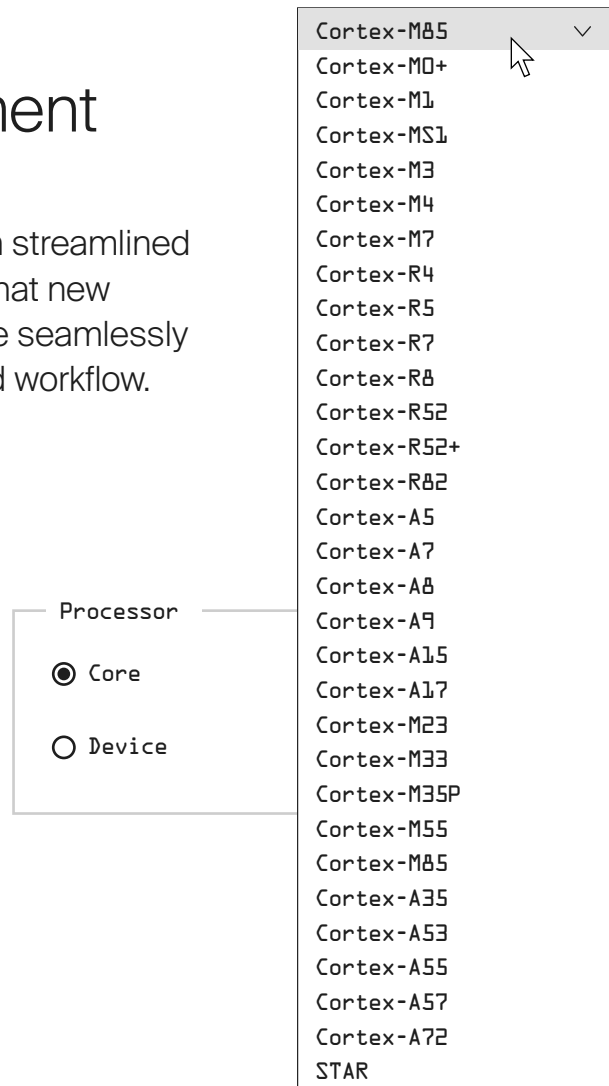
The development team's productivity relies on streamlined development processes with the assurance that new devices, middleware, and extensions integrate seamlessly into a single toolbox enabling an uninterrupted workflow.

Today's electronic devices demand a gradually growing number of embedded systems that often include a portfolio from 8-, 16-, 32- and 64-bit applications. At the same time, embedded applications are becoming increasingly complex and sophisticated. The hunger for new products with even more differentiating features has become so big that the time to market of a single product can be a decisive factor for the success of a whole company.

The embedded development tools from IAR support 15,000 devices, covering 8-, 16-, 32- and 64- bit MCU/MPUs from over 200 semiconductor partners, serving some 100,000 developers worldwide. This is the broadest device support and the strongest ecosystem. IAR Embedded Workbench allows customers to move freely between processors from all major vendors, in one single IDE and environment. However, each architecture still requires a separate license.

The market's broadest device support is made possible through a generic platform and common components for the different targets. Moreover, IAR adds architecture- and processor-specific adaptations and optimizations that let developers create efficient, stable code, and at the same time improve development efficiency.

Choosing a compiler that provides an integrated development environment which provides efficiency to shorten development time is paramount. And this is a key factor in ensuring consistency in application stability. Standardizing on embedded development tools like IAR Embedded Workbench gives development teams streamlined development processes, an uninterrupted workflow, and a single toolbox in which all components integrate seamlessly. It also simplifies development and



enables code reuse across projects and processors, thus avoiding any delays in production.

Why is this important? It is common for development teams to work concurrently with several different processors. They need to be able to choose the processor best suited for the application at hand. And if they for some reason need to change the processor, they do not have to start from scratch, but benefit greatly from not having to change the tools as well.

There is an initial investment, which pays off on the long run: To get started, it takes a developer at least one working week to learn a new IDE and toolchain. In-person training courses like "Getting Started with IAR Embedded Workbench", "Efficient Programming" and "Advanced Debugging" from the [IAR EMB Academy](#) which help developers to fully leverage the tool suite's features require three working days.



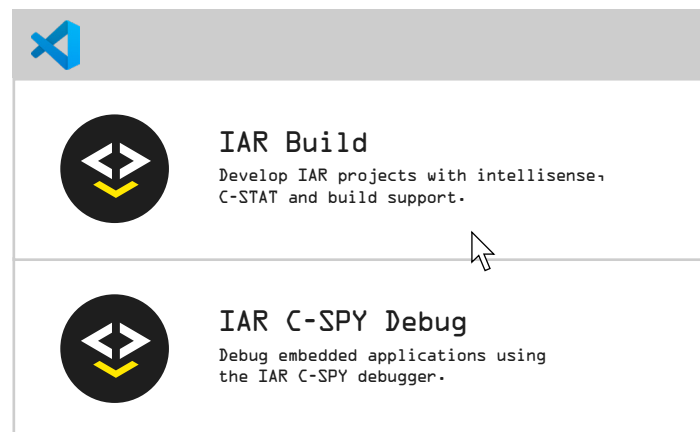
Additionally, developers need two extra days to digest the information and try the tutorials by themselves. Given that the average salary for an [embedded developer in the US](#) is \$104K x 1.4 (cost of salary including the benefits, payroll taxes and corporate liability insurance), this translates into a cost of almost \$2,800 per developer to get up to speed with the new toolchain.

This does not take into consideration that it might take way longer for the developer to get confident in the new toolchain so the consistency in application and stability can be secured. But these one-time expenses for the training of developers are a worthwhile investment – which does not have to be made again just because a different processor is used. Especially as the developers' growing experience and expertise in working with a familiar toolchain leads to further time and cost savings.

Modern development workflows also demand flexibility and additional integrations with solutions from partners in the ecosystem. The IAR Visual Studio Code extensions are compatible with all the latest versions of IAR Embedded Workbench and IAR Build Tools and are available at [Visual Studio Code Marketplace](#) enabling developers to build and develop projects from VS Code. The same applies to the IAR Eclipse plugins to take direct advantage of the high-quality IAR build toolchain alongside the advanced features. The extensions can be used for other build systems, such as CMake, source control and versioning extensions like GitHub to meet the development demands.

The IAR Embedded Workbench and IAR Build Tools include access to professional technical support for the first 12 months and access to the IAR Academy on-demand courses for a smooth start and improved productivity.

The all-in-one integrated IDE and extensions enable programmers to have the same set of capabilities in one place without needing to constantly switch tools. Tighter integration of development tasks boost developer productivity and efficiency.



Test your code for different MCU/MPUs and with VS Code Extensions by downloading the IAR Embedded Workbench

Download →

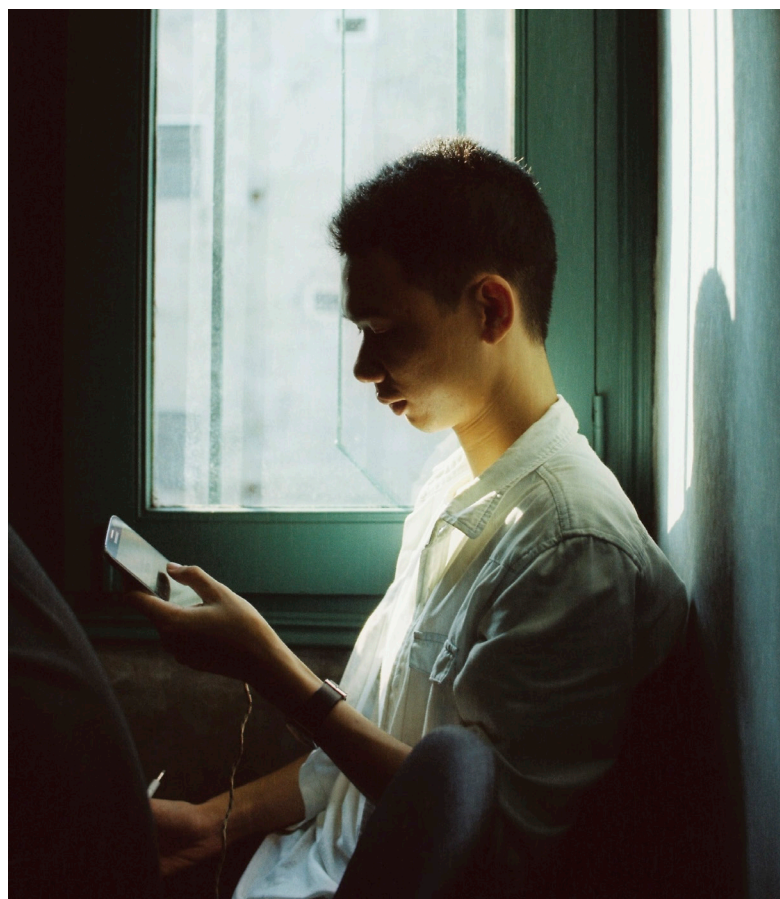
8. Compliance & Safety

Speeding the path to safety certification with certified tools. Functional safety is highly desirable in any application – but for some applications, it is an absolute requirement. Building applications with ensured functional safety can be both challenging and time-consuming unless you choose to work with pre-certified development solutions.

There are numerous benefits to following the safety certifications: It will reduce liability risks associated with your application and reduce the odds of product recall and the number of firmware updates. Additionally, it will ensure compliance with international standards and requirements aside from protecting your company's reputation and the corporate objective.

There are many standards and safety certifications in place. Each one caters to a specific industry or product category. The two most broad-reaching certification standards are ISO 26262 (road vehicles) and IEC 61508 (electronic safety-related systems), which is considered the umbrella of the certifications. Most functional safety development tools aim for certifications according to these two standards because they cover almost all other certifications and industries.

Specific certifications may go above and beyond these two standards, but they are considered the basis for many others, e.g., the EN 50128 for railway systems which is similar to IEC 61508. In general, all standards provide clear processes to assess risk for safety critical systems and assign safety goals. Additionally, also covered are the best practice development process requirements in order to reduce systematic failures. Finally, there are also ongoing procedures to ensure functional safety after product deployment. In short: The standards outline how to identify and deal with risks, and all of them require tools certified for functional safety.



Broad coverage of safety standards



Industrial
IEC 61508



Machinery control
ISO 13849 IEC 62061



Railway
EN 50128 EN 50657



Medical
IEC 62304



Agriculture & forestry
ISO 25119



Automotive
ISO 26262



Process industry
IEC 61511



Household appliances
IEC 60730

Functional safety certification for a tool means that the development tool has gone through a rigorous qualification process to ensure that it produces reliable and repeatable results when compiling code. Additionally, it means that development processes are in place to manage how the tool works with specific requirements put forth by different functional safety standards and there are test and quality measures of the tool that show validation of compliance with different language standards.

The certification process is rather rigorous. The IEC 61508 standard details how support tools should be qualified in Section 7.4.4, but it is rather ambiguous on how a compiler should be qualified. Consider clause 7.4.4.10:

“The selected programming language shall have a translator which has been assessed for fitness for purpose including, where appropriate, assessment against the international or national standards.”

These and other stipulations make it difficult to certify a tool on your own and can result in significant work on your part to prove fitness and even more work to document why you think you have proven fitness. This only gets worse as you try to achieve increasingly higher Safety Integrity Levels (SIL).

Part of the process is running a set of validation test suites in which thousands of test programs are compiled, and the results are compared against expected results. Another part is the standard-conformance tests. None of these tests are exhaustive but should identify some issues.

The trick for safety validation is to make sure all known issues are documented. Functional safety means that the known imperfections are clearly documented and that you have a process in place to identify and document imperfections. For full validation you must fix or document and justify each and every deviation from expected behavior, so there are no known unjustified deviations.



For continuous advice on how to improve developer efficiency, follow IAR Embedded Development on LinkedIn →

Validating your own toolchain for functional safety is expensive and time-consuming. Tool certifications can take up to 12 months and occupy several employees, nominally two to four. Considering the salary for an embedded developer in the US, the estimated cost can be up to \$145K depending on the extra testing requirements. The actual numbers will inevitably depend on which SIL your project requires.

Notice that if you want to reuse an uncertified tool from another project that did eventually achieve certification, then you will still be required to prove that your new project is similar enough to the previous one. You have to provide evidence that you use the same functionality of the toolchain as for the previous project that is impossible without source code-level access. In addition, you must prove that you use the toolchain in an equivalent manner like the one with safety certification. Usually, you might end up having to do the same work to requalify the tool.

By using an already functional safety-certified development tool, you no longer have to prove your toolchain complies with the safety standard – you only need to certify your application. In fact, using a functional safety-certified toolchain and coding standards can save a lot of time and money as it eases and speeds up the application certification process. It also means that the test-and-fix phase of the Software Development Lifecycle (SDLC) can focus on finding bugs in the source code instead of wondering if a compiler issue is causing problems.

The IAR Functional Safety solutions include tools certified by TÜV SÜD covering 10 different safety standards with long-term support through a special functional safety agreement and safety certificate renewal for the duration of the agreement. For customers working on safety-critical software, IAR offers prioritized support with the Functional Safety SUA, offering a response time of only one day, and a repair time for critical issues in no more than 10 working days:

Response and repair times within functional safety SUA

Source: IAR

Security level	Response time	Repair time
Critical	1 working day	No more than 10 working days
Serious	1 working day	No more than 20 working days
Moderate	1 working day	At next scheduled update or upgrade of the product, but not later than one year
Minor	1 working day	At IAR's discretion

Assess the complete development environment for Safety by downloading the standard IAR Embedded Workbench





Download →

9. Licensing

Finding the right license type can maximize the ROI. There are many use cases for licenses, and the correct mix and management of licenses essentially help optimize your tools' spending. It all comes back to the question of what the organization or development team needs.

License types

[Learn more →](#)

 <p>Stand-alone</p> <p>— Locked to One computer</p> <p>— Users Single developer</p>	 <p>Mobile</p> <p>— Locked to USB Dongle</p> <p>— Users Single developer, multiple computers</p>	 <p>Network</p> <p>— Locked to One site</p> <p>— Users Local teams</p>	 <p>Global</p> <p>— Locked to Multiple sites</p> <p>— Users Global teams</p>
---	--	--	--

IAR offers flexible licensing and pricing options to maximize the return on investment for companies. License types from standalone, mobile to network and global licenses enable easy management of licenses, and license pools are displayed above.

The Network License is convenient and cost-efficient for a team of developers. It allows sharing of a pool of licenses among a group of users over a local network. While there is a limit on the number of concurrent users, the number of installed copies that can occupy a license is unlimited. The Network License is managed by a license server that is included in the delivery. New users can be added to an existing Network License.

For customers with operations and development projects at several sites and in different countries, IAR offers geographical flexibility through Global Network Licenses. The ordinary Network License is restricted to one single geographical site, whereas the Global Network License provides the possibility to have users accessing the same network license from multiple sites globally.

A Mobile License is a single-user license that allows you to be flexible with your work location. It is locked to a USB dongle that you can bring with you and use with any PC anywhere. It works even if the PC is without a network connection. Keeping the license on a dongle also protects your license from hardware failure.



A PC-Locked License (Stand Alone) is locked to a specific PC. It is a personal, single-user license and can only be used by having physical access to your PC. It also works if the PC is not connected to a network. Let us examine a hypothetical company with two sites having 30 developers that need access to the development tools, 15 in each site, and a project duration of 2 years (20% yearly for Support and Update Agreement renewals).

Considering the standalone license being the price reference (\$) that could be any local currency and depends on the architecture (8-,16-,32- and 64-bit). The other license types have a premium cost depending on the flexibility. The dongle license costs 16% extra (1.16\$), the network license costs 33% extra (1.33\$) and the global license costs 100% extra (2\$).

Providing standalone licenses to all developers would cost $30 \times \$ = 30\$$ (34.8\$ in the case of dongle licenses). If the company would move to network licenses that cost would be, 10 network licenses (recommended) covering the 15 developers on each site. Network licenses are only allowed for the local site resulting in a cost of $10 \times (1.33\$) = 13.3\$$ per site, totalizing 26.6\$ with a cost reduction of ~13% compared to the standalone model. The next step would be moving to global licenses and that would mean 12 licenses total (recommended but could be less if the sites don't have overlapping working hours) for both sites with a cost of $12 \times (2\$) = 24\$$. This is a cost reduction of ~11% compared to network licenses and ~25% compared to standalone licenses. This example is not considering the administration and problems with damaged workstations or lost dongles that need to be replaced. Notice that the 20% for the yearly Support and Update Agreement will also follow the cost reduction percentages.

Finding the right license type for your organization can have a significant impact, reducing up to 25% of the total licensing cost but also facilitating the management of the licenses.

Experiment with the IAR flexible development environment by downloading the IAR Embedded Workbench

Download →

10. Conclusion

Keeping control of a product's development time is critical to controlling costs and meeting delivery targets. These goals can be achieved using tools and services designed to help engineers create products quickly and efficiently.



Using commercial tools with an upfront cost versus “free” tools offered to lower barriers to entry for using specific chips in product designs can be an effective way to stay on schedule and reduce the overall cost of developing a product. The companies and development teams are “buying” a faster time to market and securing the delivery of products with quality. Finally, the use cases provide a clear picture on the return on investment (ROI) and total cost of ownership (TCO) for developers working with professional solutions, like the ones from IAR.

Summary of ROI and TCO

Source: IAR

Use case	Saving & improvements	Savings
Why should you care about code size and benchmarks?	\$1 to \$4 per device	\$10K to \$40K per product series
How can the applications performance influence your BOM (bill of materials)?	\$0.50 to \$0.90 per device	\$5K to \$9K per product series
Lowering compilation time to increase productivity	increase productivity by reducing build times up to 50% (Linux)	Save up to 50% instances/hours in cloud services
Shorten debugging time for faster time to market	Free up person hours ~1.000 hours	\$24K per year
The cost of defects and why code quality matters	\$1.9K per 1.000 LOC	\$19K to \$95K per 1.000 to 5.000 KLOC
Access to technical support pays off your development tools	\$2.4K per month	\$43K per project
Broadest device support in one IDE improving developer's productivity	\$2.0K per developer	\$10K per project
Speeding the path to safety certification with certified tools	Up to \$145K per project	\$145K per project
Find the lincense model that fits your needs	Maximize usage and secure investment, depends on number of developers	Save up to 25% of the total licensing cost

Disclaimer: The information and numbers in this report are approximations for general informational purposes only and will be updated quarterly. The numbers and results might vary from version to version. IAR makes no representation or warranty, expressed, or implied. Your use of the information and interpretation of the results of the use cases is solely your own responsibility. This report may contain links to third-party references and content, which we do not warrant, endorse, or assume liability for.



For continuous advice on how to improve developer efficiency, follow IAR Embedded Development on LinkedIn →

Authors

Author



Rafael Taubinger
Sr. Product Marketing Manager

Contributors



Anders Holmberg
Chief Technology Officer



David Källberg
FAE Manager EMEA



Shawn Prestridge
FAE Manager US



Hyun-Do Lee
Sales Manager

What's next?

Speed up time to market and secure quality with powerful integrated solutions by downloading the IAR Embedded Workbench

Download →



iar.com

Tomorrow's intelligence,
delivered today

