Next-Generation IDE: Maximizing IP Reuse

Atmel White Paper

Author:
Joerg Bertholdt, Director of Marketing, MCU Tools and Software, Atmel Corporation

Abstract

The integrated development environment (IDE) has gradually changed the face of software development over the past couple of decades. Providing a single environment for writing code, debugging and deploying code to the target, the IDE has made development teams more productive than ever. For many embedded systems developers, however, there remains a gap that impacts the design cycle.

There is often a disconnect between the IDE and the documentation and applications support needed to combine application code with drivers and middleware functions to create the target system. This is not only encountered when creating new applications but also when taking an existing design and scaling it to a different microcontroller to meet market demands for more features or lower cost. To take advantage of the new target device, the software development team has to learn about a completely new set of drivers and firmware, possibly in addition to changes in core architecture. The time it takes to get up to speed on these changes adds unacceptable, and unnecessary, delays to the project.

Market surveys have indicated that the majority of embedded systems developers want to be able to reuse their intellectual property (IP) in the context of a familiar software environment. In the most recent Embedded Systems Design survey published by UBM Electronics, more than half of all projects currently undertaken by participants were upgrades or improvements being made to earlier or existing systems, with the majority of participants saying they had to add new software features. The second most common reason for changing or upgrading was for deployment on a new or different processor. As such, software compatibility across embedded platform architectures is a key requirement to maximize IP reuse and minimize the time it takes to gain experience with the new platform.

Atmel has responded to this demand by streamlining IP reuse and software scalability to further evolve the embedded systems IDE. The result is a carefully architected integrated software framework that plugs seamlessly into the IDE, maximizing IP reuse and expanding the scalability of both existing and new code across a wide range of embedded platforms.

Atmel Studio 6 also includes the Atmel Software Framework (ASF) which works in concert with the IDE. ASF is a collection of production-ready source code such as drivers, communication stacks, graphic services and touch functionality with over 1,100 project examples with source code to accelerate development of new applications.

Designers can download the Atmel Studio IDE, which includes ASF from Atmel's website, free of charge, at: www.atmel.com/atmelstudio-whitepaper.

The Evolution of the IDE

Software plays an increasingly dominant role in the success of the end product as the processing capability of embedded systems increases. Over the past couple of decades, the availability, flexibility and capabilities of IDEs have improved in response (Figure 1).

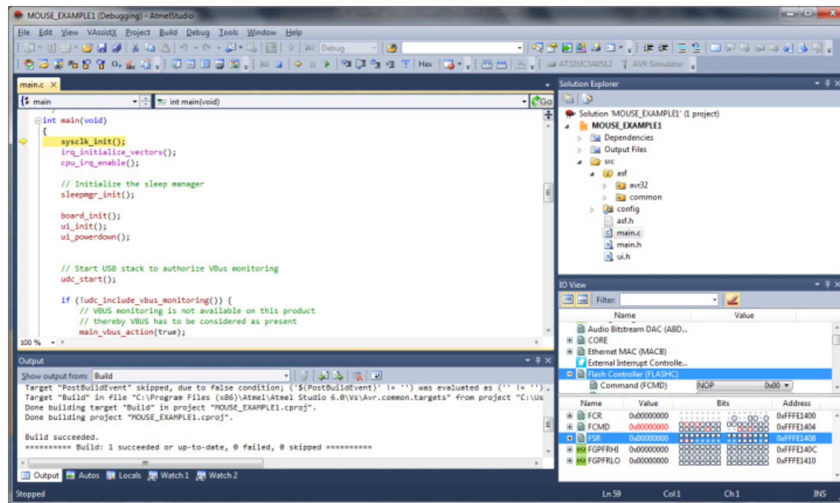

Figure 1: Professional IDE for embedded development: Atmel Studio 6

The tight integration of the environment improves programmer efficiency by providing analysis of the code before it is sent for compilation. For example, common syntax errors can be highlighted to reduce the probability of a compile step failure or to provide text expansions to reduce typing overhead. Auto-completion is available for symbols that have been defined within the project, and tool tips can show how variables within functions should be used.

Developers working on desktop- or server-focused software have had the benefit of IDEs with increasingly effective productivity enhancements. Software frameworks are now commonplace in the desktop development. They have relieved programmers of the burden of continually reinventing the wheel, writing the same algorithms and functions time and again in different projects because that was easier than attempting to edit old code to suit a new project.

However, realizing the advantages of those frameworks has taken some time. Developers found that it was not practical to commit to memory more than a few operating-system functions and classes at any time. Sometimes, they would find it faster to re-implement many of the functions than to pore through large reference manuals to see if an appropriate class existed and to determine the way in which code should access it.

Desktop-oriented IDE providers responded by incorporating framework awareness into their products. The IDEs provided code editors that were aware not just of core language syntax but of the application programming interface (API) calls supported by the software frameworks. This helped to deliver on the promise of faster, lower cost programming by giving programmers context-sensitive hints on parameters and functions supported by the framework (Figure 2).

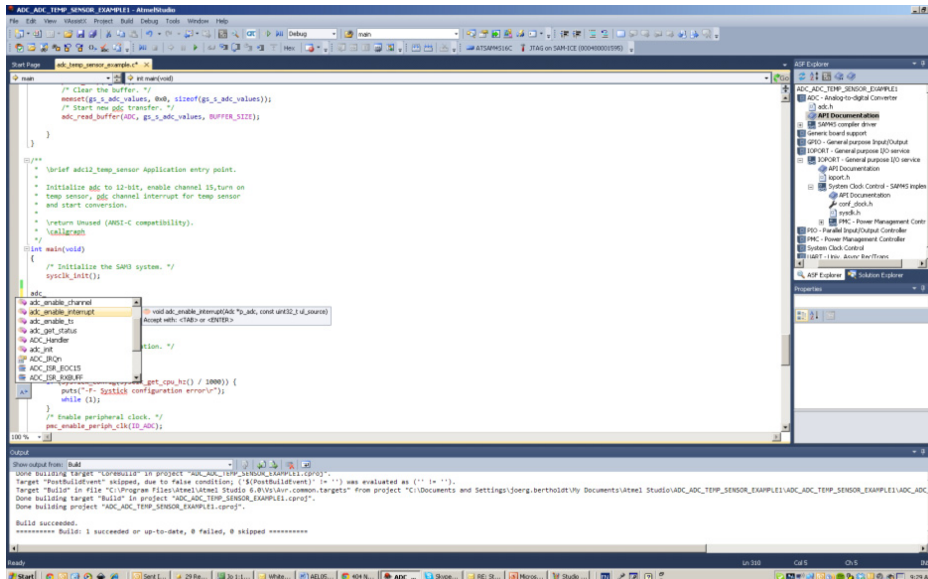

Figure 2: Context aware code editing in Atmel Studio 6

As MCU vendors have integrated more functions into their devices, embedded developers have experienced similar issues with code reuse. Documentation and application support for embedded platforms and high-integration MCUs are frequently provided purely through downloadable archives. These archives may take the form of ZIP files that contain an entire body of source for an individual MCU or family of peripherals. The documentation is often simply a digitized product reference manual – stored as a PDF that contains an index but with rudimentary search capabilities. In order to kick-start a project, it becomes the developer's responsibility to locate and download the relevant code and documentation archives and then assemble them for reference. Integration into the IDE may be little more than providing a path to the suppliers' source files so that the compiler and source-code control tools can locate them when necessary.

To consult the documentation, the developer has to flip frequently back and forth between the IDE and a PDF-display tool. This is satisfactory for the situation where a developer is working intensely on a specific function. But, during debugging, the programmer may need to consult many different parts of the product reference material to ensure that functions are being called properly and that the code is accessing registers in a correct manner. For each different function, the programmer has to scroll to the correct page or bookmark, read and then move back to the IDE.

Requirement for Scalability

A lack of commonality between MCU products in the way that individual peripherals are supported increases the burden on the developer. Moving to each new MCU may involve downloading new support code archives and documentation and learning new APIs. Code that would otherwise be identical still needs rewriting because of changes in the API between MCU versions. This is problematic in a market environment where the ability to scale and tune offerings to react swiftly to changes in demand is vital.

For example, the organization may start with a focused product offering based around a comparatively simple MCU and a small set of sensors. Customers may demand that the product works in a networked environment, necessitating the move to an MCU with greater processing power and with suitable communications peripherals. They need to be able to deploy the application on this platform as quickly as possible; however, this is difficult if the support code for that new platform uses different APIs and naming conventions that are buried in poorly indexed electronic datasheets.

The ability to scale for cost is just as important. Marketing may identify a niche in the market where, by removing some features from the initial platform and cutting pricing for that version, it is possible to massively expand the product's reach. Engineering may find that a more cost-effective approach is possible using an 8-bit platform rather than the 32-bit target used for the original project. In most situations, this will require a large modification of the application code to use the driver and software infrastructure for the more cost-optimized platform.

This development overhead is, however, unnecessary. By taking the approach that the IDE can support an entire development ecosystem rather than just an individual project, it is possible to provide embedded systems designers with a solution that provides them with much greater scalability.

Most of today's IDEs are, by and large, project focused. What is needed is an environment that is focused on the needs of the development organization – providing the support needed to enable complete reuse. This is what the Atmel® Software Framework in concert with Atmel Studio 6 provides (Figure 3).

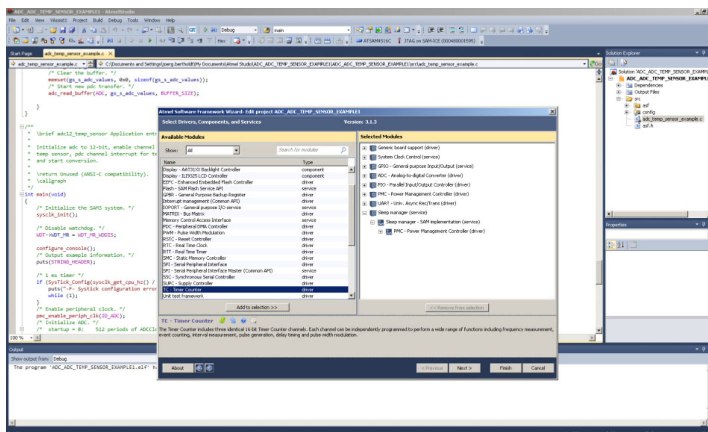

Figure 3: Atmel Software Framework tightly integrated in the Studio 6 development environment.

The Atmel Software Framework

The Atmel Software Framework (ASF) facilitates a top-down design process in concert with the Atmel Studio 6 IDE. Together, the Atmel Studio 6 IDE and ASF enable an approach to embedded systems development that fully leverages the accumulated IP and expertise of the organization by avoiding the requirement to rewrite significant portions of the code for each port to a different MCU variant or architecture. The structure of ASF means that developers can focus most of their design time on the application, not on the support environment (Figure 4).

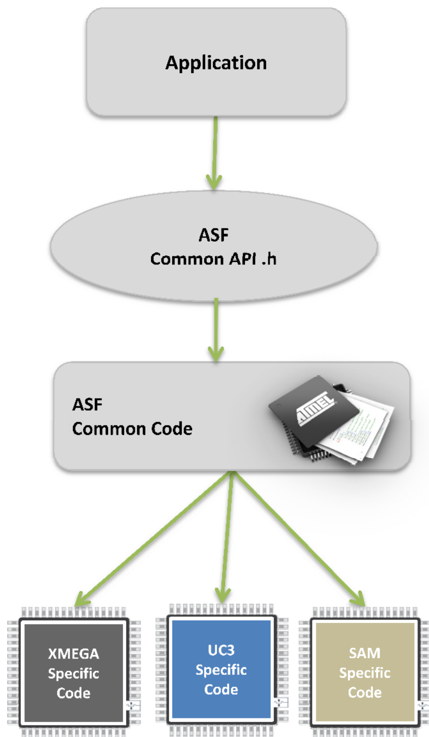

Figure 4. A common ASF architectural design

ASF has been architected from the ground-up to provide a clean interface between user application code and the software stacks that enable the application to run on a variety of different embedded target MCUs. ASF provides everything required between the application and the hardware design.

ASF is production-ready. The functions and code examples that use them are all optimized for code size for each target architecture and work with a range of ANSI-C compilers. The ASF code is also architecture-optimized by Atmel experts, ensuring not just high performance but low power. The functions take full advantage of Atmel MCU features such as low-power modes and the Peripheral Event System. Many of the drivers are interrupt-driven to avoid the power and

performance overhead of polling. And block data transfers use hardware direct memory access (DMA) features.

Chip-specific features in the protocol stacks and functions are used in a way that maximizes portability for application-level code. The functions are implemented using a common API that abstracts away the target-specific details, allowing developers to take code developed on one Atmel device and compile it for a new Atmel target practically unchanged. This portability across not just MCUs but architectures comes through the design of the ASF.

For example, the API uses an intuitive format in which devices are programmed using function calls that follow a consistent naming convention: <device>_init(); <device>_enable(); <device>_disable(); <device>_start(); <device>_stop(); <device>_read(); <device>_write().

ASF Architecture

ASF modules are arranged in a layered architecture that allows application code to call functions that are most appropriate to the task at hand. This architecture also makes it easy to add support for complex protocols, such as USB, to a product. There are four types of ASF modules: component, service, peripheral and board.

The component and service modules are called directly by the application, unless it needs direct access to low-level device functions provided by the peripheral and board layers.

The service layer provides the user with application-oriented software stacks such as USB class drivers, file systems, architecture-optimized digital signal processing functions and graphics libraries. This layer also takes advantage of the MCU's hardware features.

Components are high-level drivers that provide intuitive and direct control over MCU and board-level peripherals, such as Atmel DataFlash®, display, sensors and wireless interfaces. The code in the component layer is written with a focus on providing the functionality that a typical user will need in each of the on-chip peripherals. If the application calls for a peripheral to operate in a way that is not directly supported by the component layer, it is easy for the user to modify the existing source code, or add an extra function to the API.

The component and service modules communicate with low-level drivers in the peripheral layer that provide register-level control over hardware interfaces. Applications can also call these drivers directly to facilitate tight hardware integration in a way that maximizes portability between members of the Atmel MCU portfolio.

Finally, the board module provides the hardware view of the MCU in its target environment. The board code abstracts the component and services modules from the physical wiring and initialization functions that take care of I/O and external devices. The board code also identifies which board features are available to the modules higher up in the software stack.

The board definition provides a convenient way to assign digital and analog peripherals to each I/O pin. An application, such as an audio interface for a PC, may call for the allocation of a USB port, ADC and DAC channels, an SPI port and several general-purpose digital I/O channels that,

on the PCB, are connected to buttons and LEDs. The board definition makes it easy to add useful, descriptive names to the I/O channels that are then used by application and driver code. For example, an LED that shows whether the MCU is asleep or active might be defined as `GPIO4`. Instead of having to remember this name, the developer can assign the more logical name of `ACTIVITY_LED` to the bitmask that is used to access the specific bits within the actual I/O port's register that controls the LED state. Once defined, this constant can be used consistently throughout the application to provide access from I/O function calls to the LED.

As another example, when the MCU is ready to sleep, before calling the `sleepmgr_enter_sleep()` function – exactly the same function call is used in code for the AVR® UC3 and Atmel SAM families – the application can call the function `LED_Off(ACTIVITY_LED)` or `LED_Toggle(ACTIVITY_LED)`.

When the device wakes up from sleep, the programmer can show activity on the board by calling the `LED_On(ACTIVITY_LED)` or calling the toggle function again. This approach to code development greatly helps make the device control code self-documenting compared to the use of generic I/O functions that demand the programmer generate bitmasks for anonymous general-purpose I/O ports after looking up the necessary bit definitions in a datasheet. This low-level example illustrates the use of logical names within ASF – such low-level access to LED ports will usually be taken care of by code in the component layer.

In the embedded environment, function calls incur an overhead that may be acceptable on higher-end targets but damaging to performance on those with fewer processing resources. The ASF makes use of C features such as function-like macros to provide the ease of use of functions without the runtime overhead. These macros are used widely in the board-definition layer to keep its overhead to an absolute minimum. The macros are processed at compile time into inline code or to generate constants that can be inserted into the code directly. The ASF is structured such that code that is evaluated only once can be expressed as macros that remain source-code compatible with equivalent functions aimed at other devices within the Atmel family.

To ensure consistency in the way that module APIs are used, code provided by the ASF uses standard techniques for initialization and other management tasks. This helps reduce training time on new functions as programmers can expect to use API calls in a similar way to known modules when they incorporate a new function. For example, starting and stopping a module is normally performed by `module_start(…)` and `module_stop(…)` API calls. When encountering an A/D converter module, the programmer can expect to use functions of the type `adc_start(…)` and `adc_stop(…)`.

Separation of Interface and Implementation

Function-call conventions and parameters remain the same across the range of Atmel MCUs, including the Atmel AVR UC3, megaAVR®, AVR XMEGA® and the SAM Cortex™-M processor-based product lines. But the design commonality goes much further, reaching into every detail of the ASF's implementation.

ASF takes full advantage of the IDE and the structure of C code to ensure maximum applications compatibility—even across architectures, allowing common code development for 8-bit and 32-bit targets, even using different compilers. This attention to detail enables complete scalability to suit market demands as they change.

For example, differences between compilers and the way in which they interpret information in the source files are absorbed into architecture-specific header files. This structure allows ASF to support GCC and IAR compilers for both 8-bit and 32-bit AVR and ARM® processor-based MCUs. Similarly, differences between peripherals across the various MCUs in the Atmel portfolio are absorbed into header and C source files using rules developed by the ASF architects to ensure maximum commonality and portability. In drawing up the ASF, the software architects employed a series of rules and techniques that ensure common application code does not have to be changed to deal with a change in target MCU (Figure 5).

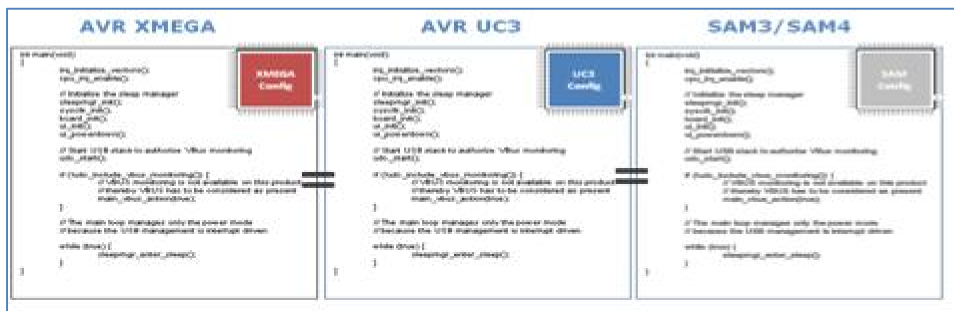

Figure 5. Identical C-code applications across Atmel's AVR, XMEGA, UC3, SAM3, SAM4 MCUs

This commonality assures ease of scaling as market demands change. It also streamlines the process of moving from prototype to production. Very often, developers will choose a more flexible, higher performance device for prototyping, so they can be assured of having sufficient headroom for the application code and potential changes during the project. As the project nears completion, it may become clear that there is potential for using a lower cost part for production – one that may even use a different core architecture. The common API employed by the Atmel devices makes it easy to port from one device to another such that the entire range of 8-bit and 32-bit targets can be viewed as a continuum of functionality and price.

Clear Coding Conventions

To improve readability as well as reusability and portability, the ASF code follows clear rules. For example, variables and function calls employ consistent naming and formatting conventions, such as all lowercase names with words separated by underscores. In contrast, constants and enumerated values, which should stand out from the rest of the code for easy identification, use all uppercase names.

This formatting proves highly valuable when sample code from the ASF is incorporated into a project. When developers start on a new project, a good way to start work is to take code from example projects and then modify it.

In a recent survey conducted by Atmel of existing ASF users, two-thirds said the example code provided in the framework proved very or extremely useful when they used it. The code is often employed in production systems. In the survey, 80 percent of users said example code was used in their product prototype.

Many of the example code segments in ASF are designed to demonstrate how functions – and their underlying hardware – can be used. The datasheet is still a reference but it is no longer the primary reference. Instead, the example code can be used as a resource for learning about the device as well as a useful starting point for implementation.

The functions in these code examples help demonstrate the tight integration between the IDE and ASF and the way in which the documentation has become an active part of the development process, instead of being consigned to a static PDF.

One possibility for bringing up a target for the first time is to select an application template. This template helps set up the basic hardware configuration, adding descriptive names to the various I/O pins and ports on the MCU. At this point, the developer can add services, components and drivers to the project using a wizard to pull in the necessary support modules. As these services and components are added, the drivers on which they rely are incorporated into the project.

Furthermore, the drivers in ASF have quick start guides that are part of their API documentation. These show and explain, in a step-by-step fashion, the code and actions needed to set up and use a driver in various use cases. The ASF employs the Doxygen code-formatting technique and engine to present the example code in a logical manner and to guide the user through the steps needed to put the code into action.

Clear Guidance Through Use Cases

For each use case, the documentation first shows how the example is expected to work. Subsequent sections in the documentation explain how to set up and use the driver. The use case may, for example, rely on a secondary driver that must be added to the project manually because there is no direct dependency between the driver and some of the example code that uses it. For example, code to demonstrate the use of interrupt-based ADC management will need to call an interrupt-management driver.

Having described how to set up the example code, the documentation shows piece-by-piece how the example code is constructed. Each step generally features a fragment of code to copy or an action to perform, together with explanatory notes to guide the user on ways to customize the software.

The tight integration between Atmel Studio 6 and the ASF makes the process of getting started with any peripheral an intuitive process. When you copy example code into a project, the IDE will check for dependencies. Functions that do not have prototypes declared within the project will not be color-coded, showing that additional header files and source code needs to be added to complete the project. The Atmel Studio 6 IDE makes this code easy to find and capture using search tools or through the help information supplied with the various examples. Once in place,

the Visual Assist tool indexes the functions and highlights the code to show that the functions and constants have valid definitions.

The example code may not have the parameters the developer wants for a particular project. And, for those just starting to work on a particular target, it is often helpful to see what parameters are available. Instead of having to scan the datasheet, the combination of example code and IDE makes this easy. For example, a quick-start guide may show that one of the first steps needed to bring up a target is to call the common function sysclk_init(), which initializes the system clock no matter which MCU is used as the target. For some applications, this is all that is needed to get the target running. However, the MCU will generally have a number of possible clock sources. The sysclk_init() function is defined in the conf_clock.h file, which the IDE will show when the user right-clicks on the function name in the source.

In that file are various defines for the clock sources for that platform, such as an internal 2MHz RC oscillator (SYSCLK_SRC_RC2MHZ) or an external oscillator (SYSCLK_SRC_XOSC). The quick-start guide will show the defines that need to be edited to make the target run at a defined frequency or ratio of the external clock; the IDE will take the user to the right place.

The example code will often have similar enumerated types in the function calls it makes. To see what other options are available for that peripheral, simply right-clicking on the enumerated type in the function – clearly delineated by the use of all capital letters in its name – and selecting the option to go to the definitions will take the user to the relevant part of the header file. This is a far quicker process than trying to download a separate datasheet and finding the relevant passage in that. For the function call itself, tooltips pop up when the user places the cursor over the function showing the variables that the function accepts and their datatypes.

Consistent Debugging

The tight integration and ease of use of Atmel Studio 6 and ASF continue into the debugging support. The plug-in model makes it possible to support a range of debugging options including a comprehensive simulator that lets developers work on code even without a hardware target.

The simulator not only runs all the instructions of the target MCU but also provides access to the peripherals. The developer can see the effect of the I/Os by modifying registers directly from the GUI or by providing stimuli to drive the virtual target. Clear highlighting shows immediately how I/O ports are affected. Register values that have changed since the last breakpoint are displayed in red with a solid red box indicating a modification from 0 to 1. A red outline indicates the opposite change, from 1 to 0. This shows clearly and quickly how the code has affected various peripherals. To see the impact of a change, simply clicking in the bit fields of a register will change its state, and the user can see how that ripples through the rest of the application.

If the developer has more than one type of debugger connected, the appropriate one can be selected in the debug tool window tab, making it easy to switch between simulation and in-circuit emulation or JTAG debugger.

In addition to extensive development and debugging support, the ASF includes a complete test framework and ready-to-run unit tests for the various peripherals and functions supported by the Atmel family of MCUs. For example, unit tests are available to perform tasks such as memory checks to ensure the integrity of Flash memory or that the USB interfaces are working as expected.

Additional Software Packages

Other features in Atmel Studio 6 include QTouch® Composer, which seamlessly ties together the tools required to tune the design of touch-based applications, making it easier than ever to integrate the increasingly advanced capabilities of capacitive touch. The software tool is designed to enable users without an expert level of understanding of touch technologies to quickly and effectively integrate touch functionality into their applications. The validation wizard in the IDE helps assess design quality by reporting capacitance, noise and reference levels, as well as notifying the user of potential margin issues. With the power analyzer, users can monitor power consumption of the microcontroller in real time to see how the CPU handles the touch-sensing code to see if there are opportunities to take advantage of longer sleep periods. Users also have access to the QTouch library, a part of ASF that works with QTouch Composer (Figure 6).

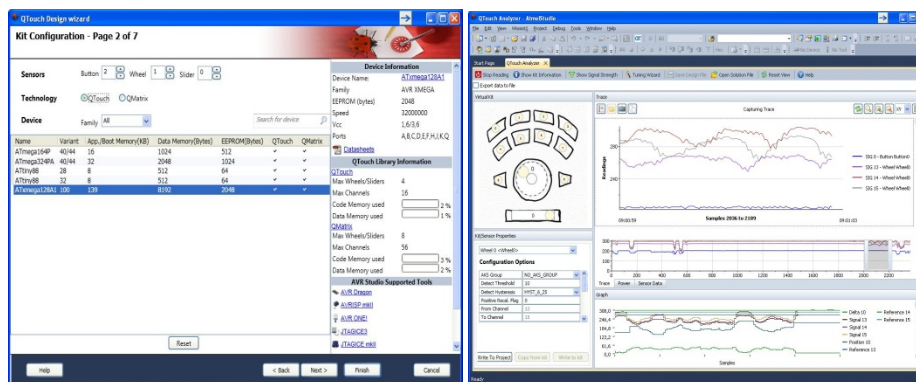

Figure 6. With Atmel's QTouch Composer integrated into Studio 6, designers can easily design touch-based applications using Atmel's AVR and ARM processor-based MCUs without toggling between the two environments.

The ease of use of ASF extends beyond Atmel's own software and support. Many third-party modules are available that hook into the ASF, including both open source and proprietary software libraries. For example, the FreeRTOS operating system is available across the range of Atmel MCUs. Other libraries include support for graphics, cryptography and wireless networking as well as the complete CMSIS library for the SAM series of MCUs based on the ARM Cortex-M architecture.

Through the combination of these many features and the tight integration between Atmel Studio 6 and ASF, the next-generation IDE is here. It's an IDE that supports the entire development

ecosystem for microcontroller-based embedded systems, not to mention the design engineer's time-to-market goals.

Summary

The combination of Atmel Studio 6 and the Atmel Software Framework (ASF) represents a clear leap forward in terms of embedded software development productivity. The tools satisfy the demands of today's embedded developer by targeting key obstacles to creating, prototyping, deploying and scaling solutions to fit the needs of the market.

Editor's Notes About Atmel Corporation

Atmel is a worldwide leader in the design and manufacture of microcontrollers, capacitive touch solutions, advanced logic, mixed-signal, nonvolatile memory and radio frequency components. Headquartered in San Jose, CA, Atmel (NASDAQ: ATML) has 40 local sales offices worldwide, as well as wafer fabrication locations in Colorado Springs, CO, and third-party foundries, Atmel is able to provide the electronics industry with complete system solutions focused on industrial, consumer, security, communications, computing, and automotive markets. In addition, the company has test and assembly facilities in the Philippines and subcontractors. For more information, visit www.atmel.com.

Atmel Corporation
1600 Technology Drive
San Jose, CA 95110
USA
Tel: (+1)(408) 441-0311
Fax: (+1)(408) 487-2600
www.atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan G.K.
16F Shin-Osaki Kangyo Bldg.
1-6-4 Osaki, Shinagawa-ku
Tokyo 141-0032
JAPAN
Tel: (+81)(3) 6417-0300
Fax: (+81)(3) 6417-0370

© 2012 Atmel Corporation. All rights reserved. / Rev.: White Paper–Studio 6

Atmel®, Atmel logo and combinations thereof, Enabling Unlimited Possibilities®, AVR®, XMEGA®, QTouch®, and others are registered trademarks or trademarks of Atmel Corporation or its subsidiaries. ARM® and Cortex™ are registered trademarks or trademarks of ARM Ltld. Other terms and product names may be trademarks of others.

Disclaimer: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN THE ATMEL TERMS AND CONDITIONS OF SALES LOCATED ON THE ATMEL WEBSITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS AND PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and products descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.