Embedded Component Based Programming with DAVE™ 3

By Mike Copeland, Infineon Technologies

Introduction

Infineon recently introduced the XMC4000 family of ARM® Cortex™-M4F processor-based MCUs for industrial real-time control and communications applications. To accompany the XMC4000 family, Infineon has also released a free, revolutionary software development tool chain called DAVE™ 3.

DAVE™ 3 has all of the features you would expect from an Integrated Development Environment (IDE), such as a code editor, compiler, debugger, etc. What is revolutionary about DAVE™ 3 is the Code Engine (CE). The Code Engine is an automatic code generator designed to enable Component Based Programming (CBP) for embedded real-time control.

This article provides an overview of DAVE™ 3, a brief introduction to CBP and how it differs from Object Oriented Programming, and demonstrates how DAVE™ 3 handles the special requirements for embedded CBP (eCBP).

Overview of DAVE™ 3

Choosing a microcontroller and software development tool chain is an important decision with lasting repercussions. For a software manager, when a software engineer brings in a new tool chain it can be similar to having your son or daughter become engaged. If they do not choose carefully, like the famous TV commercial says, you could end up having a "grandson with a dog collar".

One nice feature of DAVE™ 3 is that it can be used for the complete software development process, or you can simply use it to generate drivers or pieces of the application while continuing to use your familiar ARM® development tool chain for the rest of the design. In fact, Keil has a plug-in for DAVE™ 3 so you can convert your DAVE™ 3 project into a Keil MDK-ARM™ project with the touch of a button. Still, to ease any concerns you might have about a new tool chain, let's take a close look at DAVE™ 3 and get to know the family.

DAVE™ 3 comes from a long line of Infineon development tools. Figure 1 shows the DAVE™ Family Tree. In the late 1990's, the first version of DAVE™ was released. The first version, along with its successor DAVE™ 2.1, was primarily a code generator targeting Infineon MCU peripherals. DAVE™ 1 and 2.1 gave the users a GUI representation of the MCU and each of its peripherals. The GUI settings were used to generate C code for initialization and drivers for use during runtime. DAVE™ 1 and 2.1 could, for example, generate functions to start an ADC or transmit a CAN message.

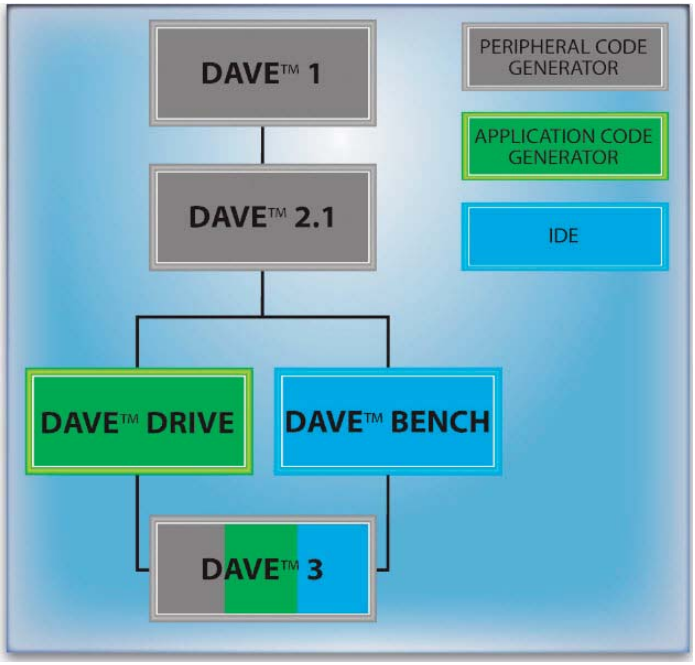

Figure 1: DAVE™ Family Tree

The code generated by DAVE™ 1 and 2.1 is easily integrated into a third party IDE project where the software developer can add higher level application code. DAVE™ 2.1 is an incredibly useful tool to get started using Infineon MCUs and to understand how the complex peripherals are configured and used. As an Application Engineer, I use DAVE™ 2.1 nearly every week to generate simple example projects to help customers get started with their projects.

For all of its power, both DAVE™ 1 and DAVE™2.1 have two main limitations: only low level code is generated and a third party IDE is required to edit, build and debug the application.

DAVE™ DRIVE was introduced to generate not only low level drivers, but higher level software for motor control applications. DAVE™ DRIVE generates low level drivers for the MCU peripherals and a higher layer of software for several different motor control algorithms. The DAVE™ DRIVE GUIs allow the user to select the type of motor, control algorithm, sensor configuration, timings, etc., to implement a fully functioning motor control application.

DAVE™ BENCH was developed to provide a free software development system for Infineon 8-bit XC800 MCUs. DAVE™ BENCH is an Eclipse-based IDE that is used to edit, compile and debug software, and works seamlessly with DAVE™ 2.1 and DAVE™ DRIVE generated code. DAVE™ BENCH has no code generation capability.

DAVE™ 3 is much more than the sum of his predecessors. DAVE™ 3's most basic features are similar to that of DAVE™ BENCH (Eclipse-based editor, C/C++ compiler, debugger, etc.). However, the Code Engine is much more powerful than DAVE™ 2.1 or DAVE™ DRIVE, and is designed to facilitate eCBP.

Component-Based Programming

Component Based Programming (sometimes referred to as Component Oriented Programming or Component Based Software Design) describes the development and partitioning of software into independent, functional, and reusable pieces. These pieces of software are referred to as "Components".

You have probably implemented some of the basic concepts of CBP without even noticing. Think of the old Hex2ASCII() (or ASCII2Hex) converter function you might have written in college and used maybe 20 times since. This function could be considered as a very basic Component. Why did you re-use the Hex2ASCII() function from college? Because it saved you time (because it is easy to reuse) and you knew that it worked (because it is independent and functional). With CBP, software becomes less complex, while development and test time are reduced.

To facilitate reusability, the interface and use of the Component must be well-defined. This allows you to consider the Component to be a black box and ignore how the Component is implemented. If you are like me, the black box analogy automatically raises red flags. Bad experiences when buying pre-compiled object code, or paying extra for source code and documentation, come to mind.

With DAVE™ 3 you do not generally need to be concerned with how the Component is implemented (i.e., you can treat it like a black box). However, the DAVE™ team has been very careful to include all the required documentation and source code to make sure – the user has the flexibility to explore and modify the Component as and when required.

Let's look at an example of how CBP could be used for a simple application and compare it to Object Oriented Programming (OOP). This simple application uses the microcontroller's analog-to-digital peripheral to read some analog signal and show that value on a display.

With CBP, we look at the application in terms of its functionality. There are many different ways that this system could be partitioned, but the three major functions in this system are:

1. Perform the analog to digital conversion and read the result
2. Convert the result to ASCII
3. Display the ASCII result on the display

Each of these three major functions could be a separate Component as shown in Figure 2a. In Figure 2a, the Components are called Apps. This is the naming convention of DAVE™ 3. Notice that these Components are described mainly by verbs. They actually do something. To make the Components reusable will take a little extra work, but we will consider this later.

The same application could look a little different when using OOP. With OOP the system is divided into data types that represent physical or conceptual objects. In C++, objects are called "classes" and contain data types, member functions, constructors/destructors and possibly even operators.

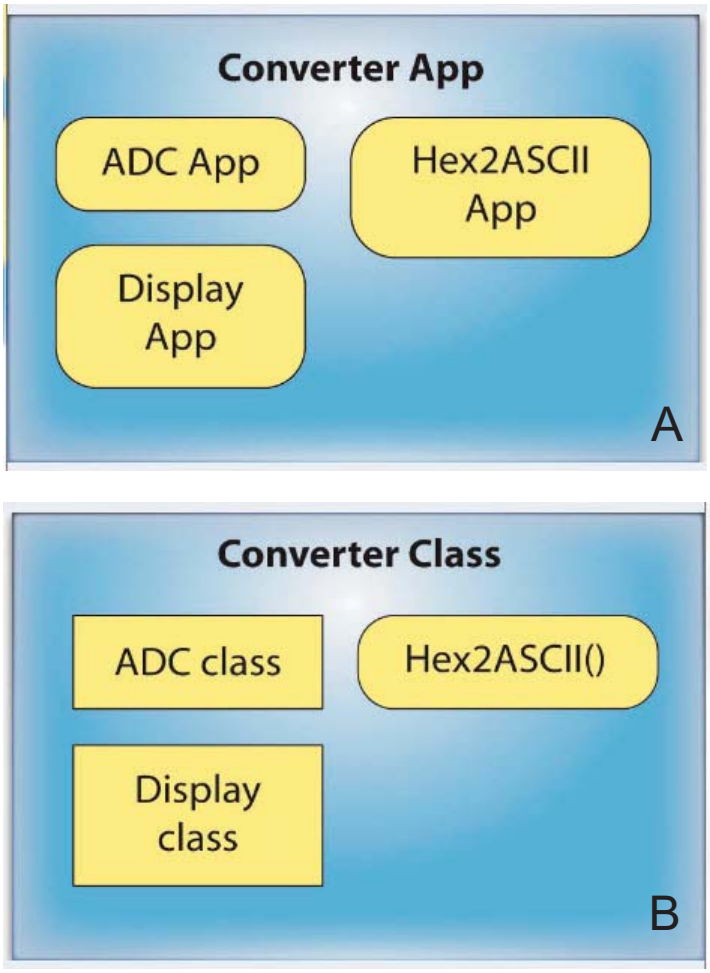

Figure 2: Partitioning of a simple application using CPB (a) and OOP (b)

Figure 2b shows how our example application could be implemented using OOP. The "ADC" and the "Display" are the two main Objects in this application. The Hex2ASCII() function could be implemented as a member function of the main Object. With OOP, the Objects are often more easily associated with nouns1. So the "ADC" class represents an actual analog to digital converter, whereas in CBP the "ADC" App represents the action of setting up the ADC peripheral and performing the conversion.

In OOP, there would be a new Object for each analog to digital converter peripheral, but in CBP there could be a new Component instance for each channel that is being converted.This does not mean that OOP and CBP are incompatible. Objects can also be Components if they are designed to independently perform specific functions.

Embedded CBP

The Hex2ASCII() function from college works so well as a reusable software component because its inputs and outputs are well-defined and because its operation is not fixed to any specific hardware. The other Components from our example in Figure 2 are not quite so simple because they must interact with hardware. This has been an impediment to using CBP in embedded systems. A significant portion of the software in embedded systems must make use of on-chip peripherals or other hardware resources. Any time software must interact with hardware, you can lose a certain amount of reusability. For example, the ADC and Display Components can easily be made to work with specific ADC channels and a specific display, but even simple changes like adjusting the ADC conversion timing or the pins that are connected to the display can require digging into the depths of the Component.

DAVE™ 3 provides a framework so Components (called DAVE™ Apps) can be written in a way that they use virtual hardware resources. For example, an ADC DAVE™ App can be written so that it can use any ADC channel of any ADC module. The App can then be reused as many times as required. The XMC4000 family of MCUs facilitates this level of Component reuse because of the degree to which the peripherals are repeated.

In the XMC4000 family most of the peripherals are repetitive in two ways. First, there are multiple instances of most peripherals (e.g., four CCU4 modules, two CCU8 modules, four ADCs, three USIC modules, etc.). Secondly, each peripheral is usually divided into repetitive channels or slices (e.g., four slices in each CCU4 module, eight channels in each ADC, two channels in each USIC, etc.).

Figure 3 shows a generalized view of a MCU of the type found in the XMC4000 family. It contains a CPU with a clock, interrupt system and peripherals. The peripherals have multiplexers on their inputs and outputs, as do the clock and interrupt systems. This creates a web of connections known as the "Connection Matrix". With the Connection Matrix, events in one peripheral can trigger actions or events in another peripheral. A simple example is triggering an analog to digital conversion and/or interrupt when a timer expires.

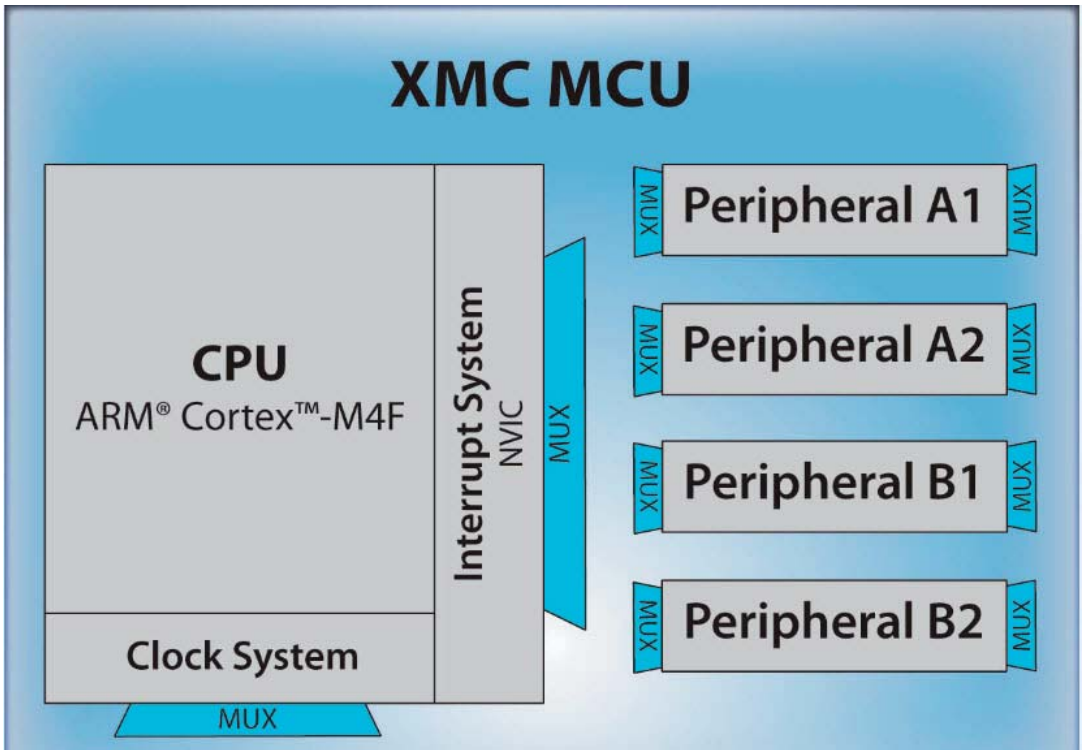

Figure 3: An MCU made up of a CPU, clock system, interrupt system and repetitive peripherals with repetitive structures

In DAVE™ 3, a DAVE™ App can be written so that it only uses the gray shaded boxes in Figure 3. This is the part of the peripheral that is repeated several times in the MCU (such as an ADC channel).

The DAVE™ mother-system uses a Device Description file that lists which peripherals and connections are available in each MCU variant. Then a special tool called the Solver is used to map the DAVE™ Apps to real peripherals and determine the right multiplexer settings.

Using DAVE™ 3 for eCBP

Instead of devoting a lot of time and space explaining the theory behind eCBP, it is much easier to look at an example of how DAVE™ 3 handles Components that require hardware resources. In DAVE™ 3, Components are called DAVE™ Apps.

Figure 4 shows a simple project that uses DAVE™ Apps to take incoming data from a UART interface and display the data on an OLED display that is connected to the MCU via a SPI interface.

When starting a new DAVE™ 3 CE project, a list of the installed DAVE™ Apps appears in the "App Selection View" window on the right (see Figure 4). The DAVE™ Apps are downloaded from the Infineon website and can be used for simple things like generating a single channel of PWM, or complex systems like a web server. Double clicking on any of the DAVE™ Apps in this window inserts them into the project. The DAVE™ Apps that are being used in the current project are shown in the lower left "App Dependency TreeView" table.

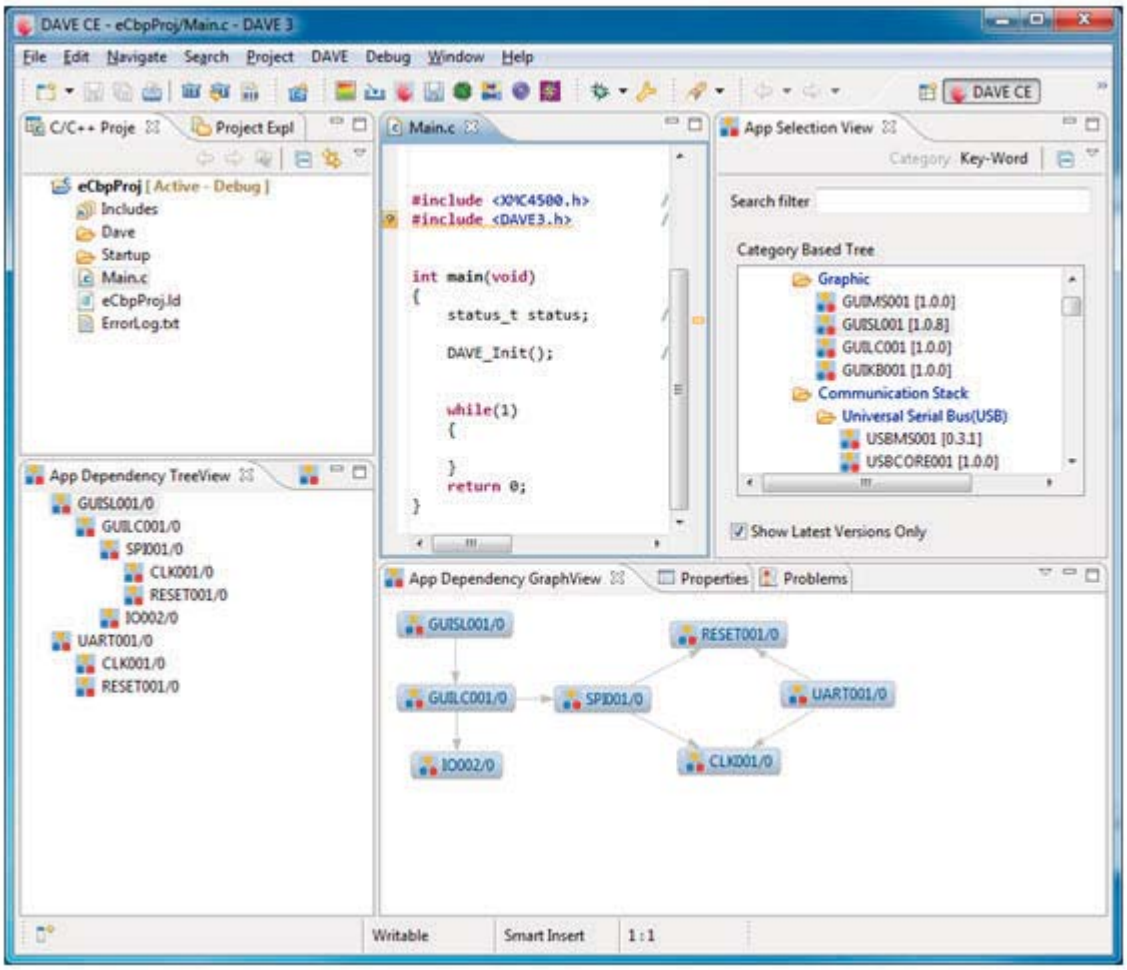

Figure 4: Simple PWM example using DAVE™ Apps for eCBP

The "App Dependency GraphView" along the bottom, shows the DAVE™ Apps that are used in the project and the virtual signals that connect them (represented by arrows). Virtual signals are a graphical representation of the connection matrix. They are called "virtual" because the Solver will assign them to actual physical signals.

This example uses a total of seven DAVE™ Apps. Only the UART (UART001) and Segger GUI (GUISL001) DAVE™ Apps were manually inserted into the project. The OLED low level driver (GUILC001), SPI (SPI001), I/O pin (IO002), reset (RESET001) and clock (CLK001) DAVE™ Apps were automatically inserted into the project since they are "required" by UART and Segger GUI DAVE™ Apps.

As you might expect, the clock DAVE™ App is used to setup the clock system. The reset DAVE™ App is used to bring the UART and SPI interface (which are channels of USIC peripherals) out of reset. These DAVE™ Apps are "singletons". This means that they can only be included once in the project because there is only one clock and reset system in the MCU. The other Apps can be included as many times as required, since the application could have many displays and serial channels.

Up until now, we have only thought of a Component as being a piece of software (e.g., C code) that runs on a MCU. However a DAVE™ App is more than just C code. A DAVE™ App can contain a User Interface (UI) that can be used to configure the App.

Figure 5 shows the UI for the UART App, containing all the standard settings you would expect.

A feature of DAVE™ Apps is the integrated documentation provided via help files. Figure 6 shows the home help page for the Segger GUI DAVE™ App. The help files include all of the information you need to understand how the DAVE™ App works, what resources it consumes and provides, API documentation, and examples of how to use the App.

At this point, if we were to run the Solver and generate code, the solver would choose which USIC modules, USIC channels and I/O pins are used for the UART and SPI interfaces. In many cases the hardware designer may have already selected the pins that should be used.

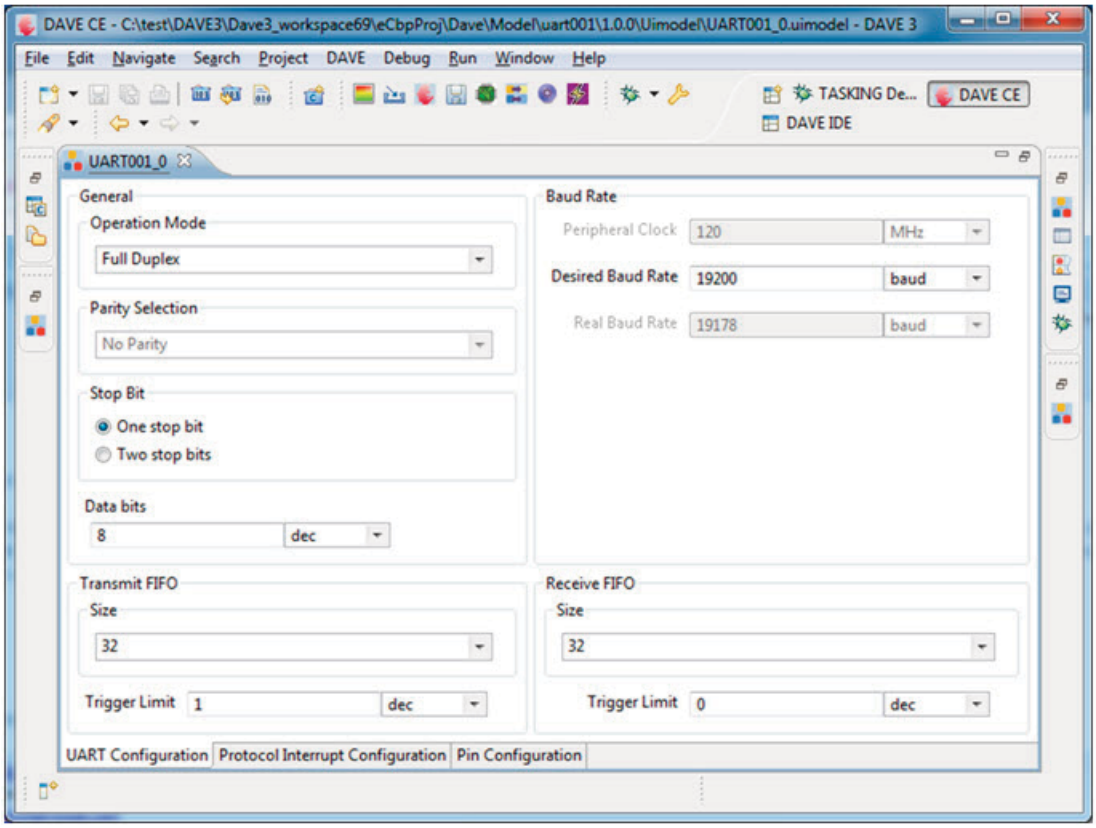

Figure 5: The User Interface (UI) of the UART DAVE™ App

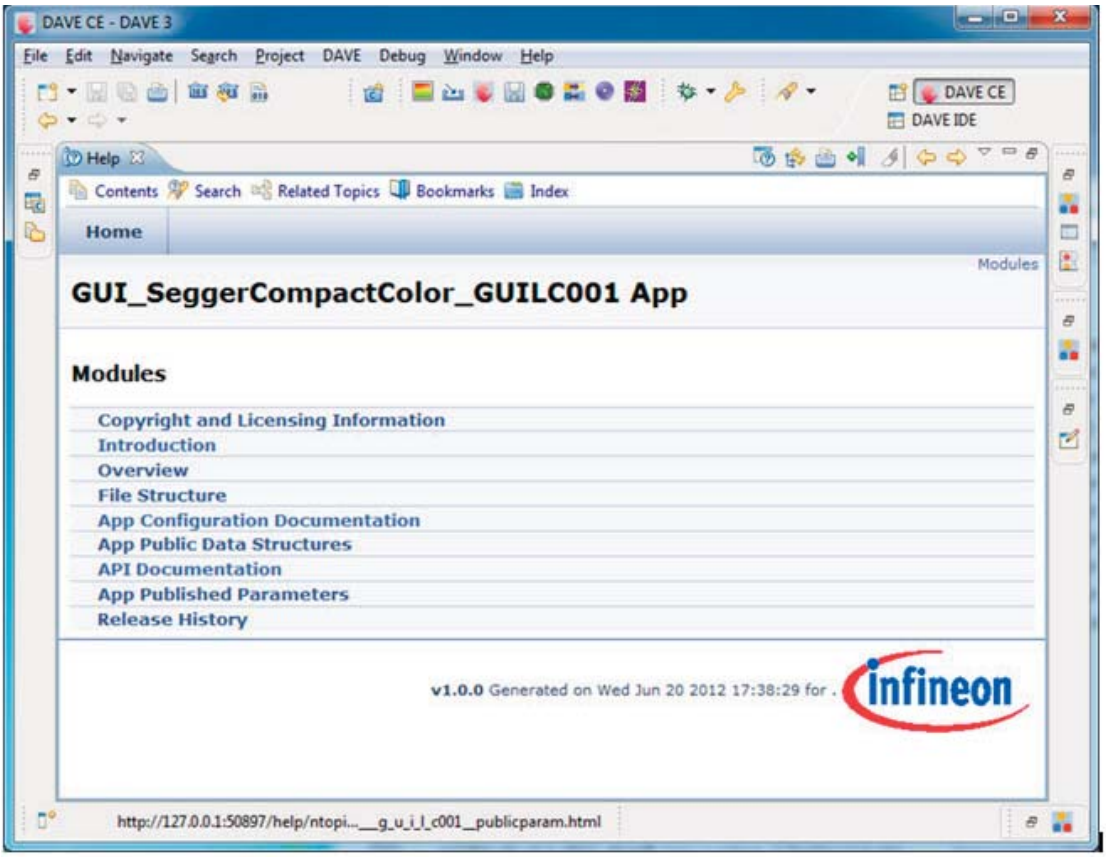

Figure 6: The home help file of the Segger GUI DAVE™ App

For example, if I want to run our UART to Display application on a starter kit, then I must select the SPI pins that are physically connected to the display IC. DAVE™ 3 allows you to do exactly this. By right clicking on the SPI DAVE™ App, you can select "Manual Pin Assignment" and force the Solver to only select SPI channels that meet these constraints.

Figure 7 shows how to constrain the SPI App to use certain pins. The actual peripherals and pins that are chosen by the solver can be viewed in the Resource Mapping Information Window and exported to .cvs file which can be opened in Excel.

The Solver works in a similar way to what you may have seen in FPGA development tools. After inserting and configuring all of the Apps that your application requires, run the Solver as it determines if all of the resources and connections that are required by the DAVE™ Apps exist on the MCU. The Solver then assigns the resources.

So far our UART to Display example includes all of the initialization and driver code that needed to setup the peripherals and transfer data. There are several options for implementing how the actual data transfer takes place.

Since the UART DAVE™ App includes a FIFO, we could periodically interrupt the CPU (e.g., every 10 ms) and then display all of the data that has been received. This could be done by inserting a System Tick (SysTick) DAVE™ App and then using the API to create a timer call back function.

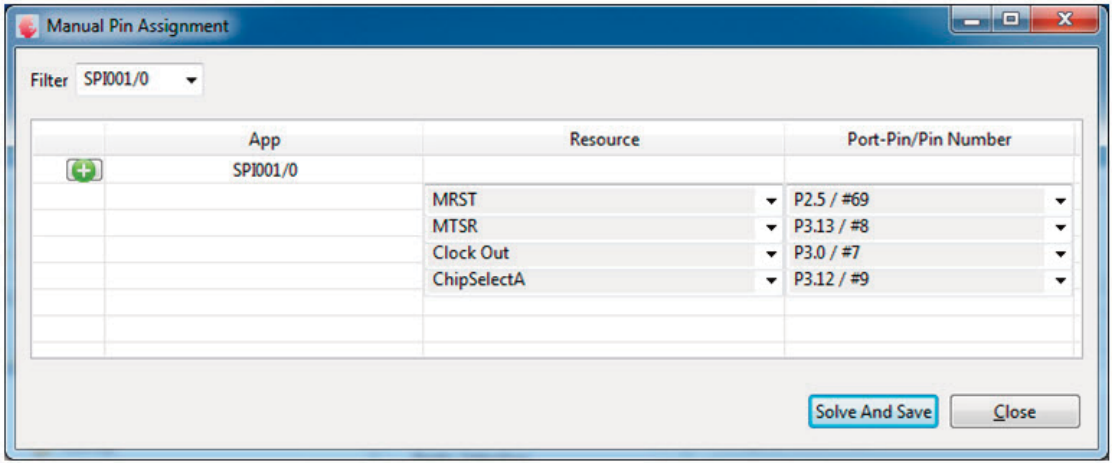

Figure 7: Manual Pin Assignment window for the SPI DAVE™ App

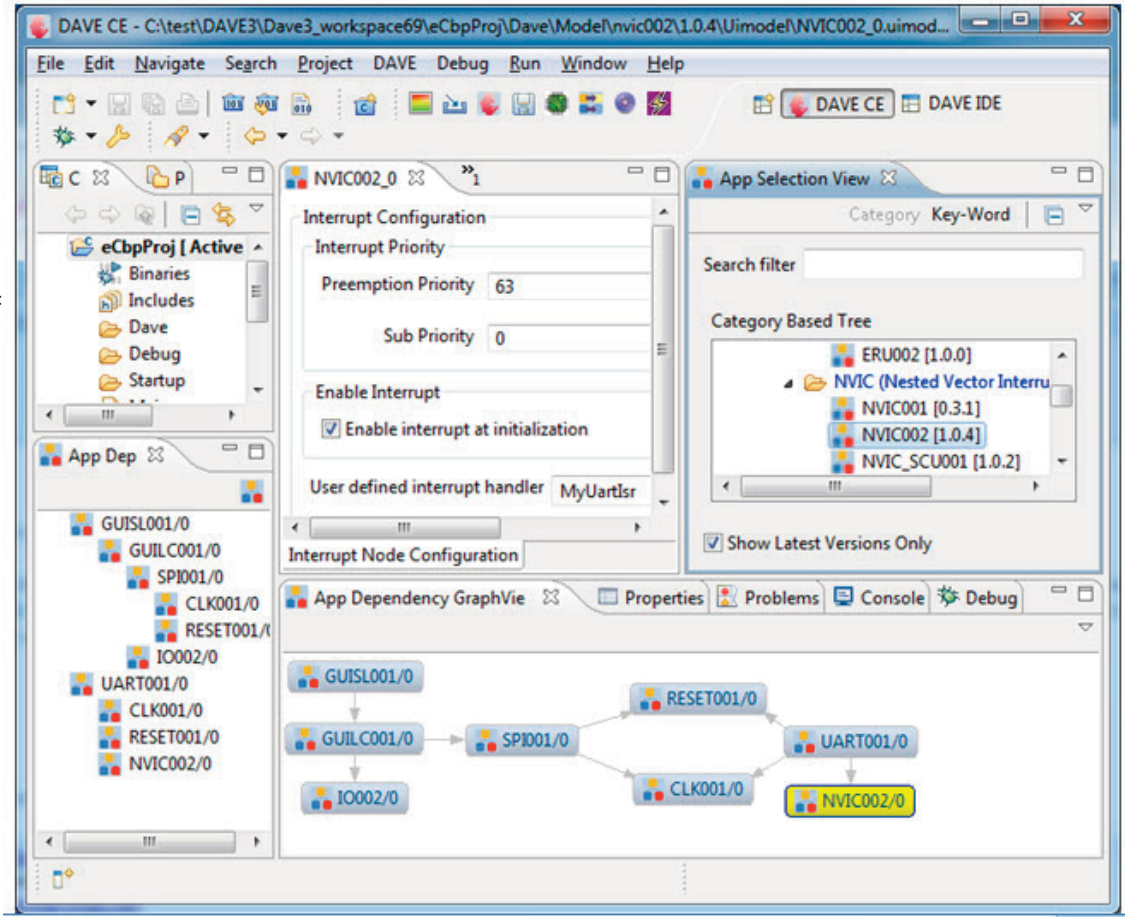

Figure 8: Inserting a NVIC DAVE™ App will setup the interrupt and insert the location of the ISR (MyUartIsr) into the vector table

Another option would be to insert a DAVE™ RTOS App (e.g., Keil/ARM RTX) and setup a periodic thread that checks the UART FIFO and transmits any data that has been received. This would probably be the most common implementation.

The easiest way to handle this simple application would be to interrupt the CPU every time a byte is received on the UART, and then display the byte via the GUI API. This is accomplished by inserting a Nested Vectored Interrupt Controller DAVE™ App (NVIC002) into the project and connecting the UART "FIFO Receive Buffer Interrupt" virtual signal to it. Then we would simply call the UART and GUI APIs inside of the Interrupt Service Routine (ISR). Figures 8, 9 and 10 show these steps.

Notice that in the ISR (MyUartIsr) the function UART001_ReadMultiple is called. This function is provided in the UART DAVE™ App API.

It is fully documented and the source code included.

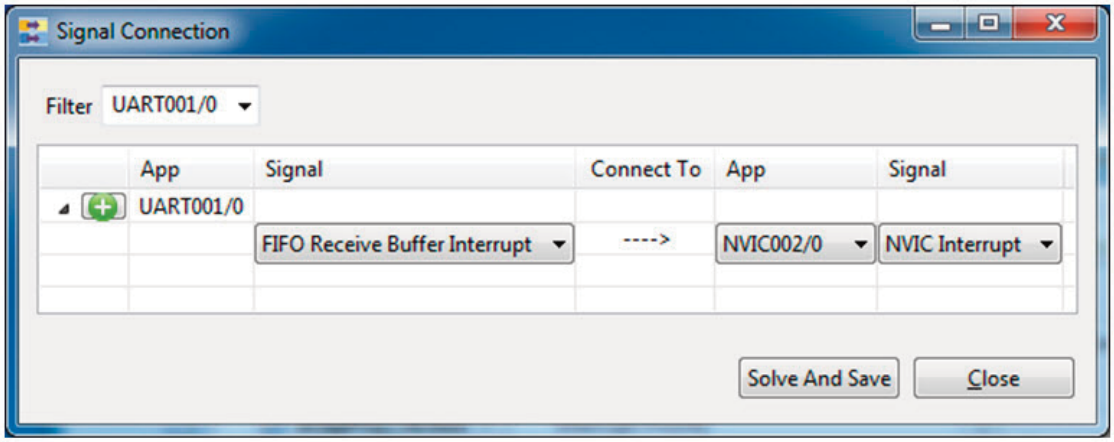

Figure 9: Connecting the UART FIFO Receive Buffer Interrupt virtual signal to the NVIC DAVE™ App

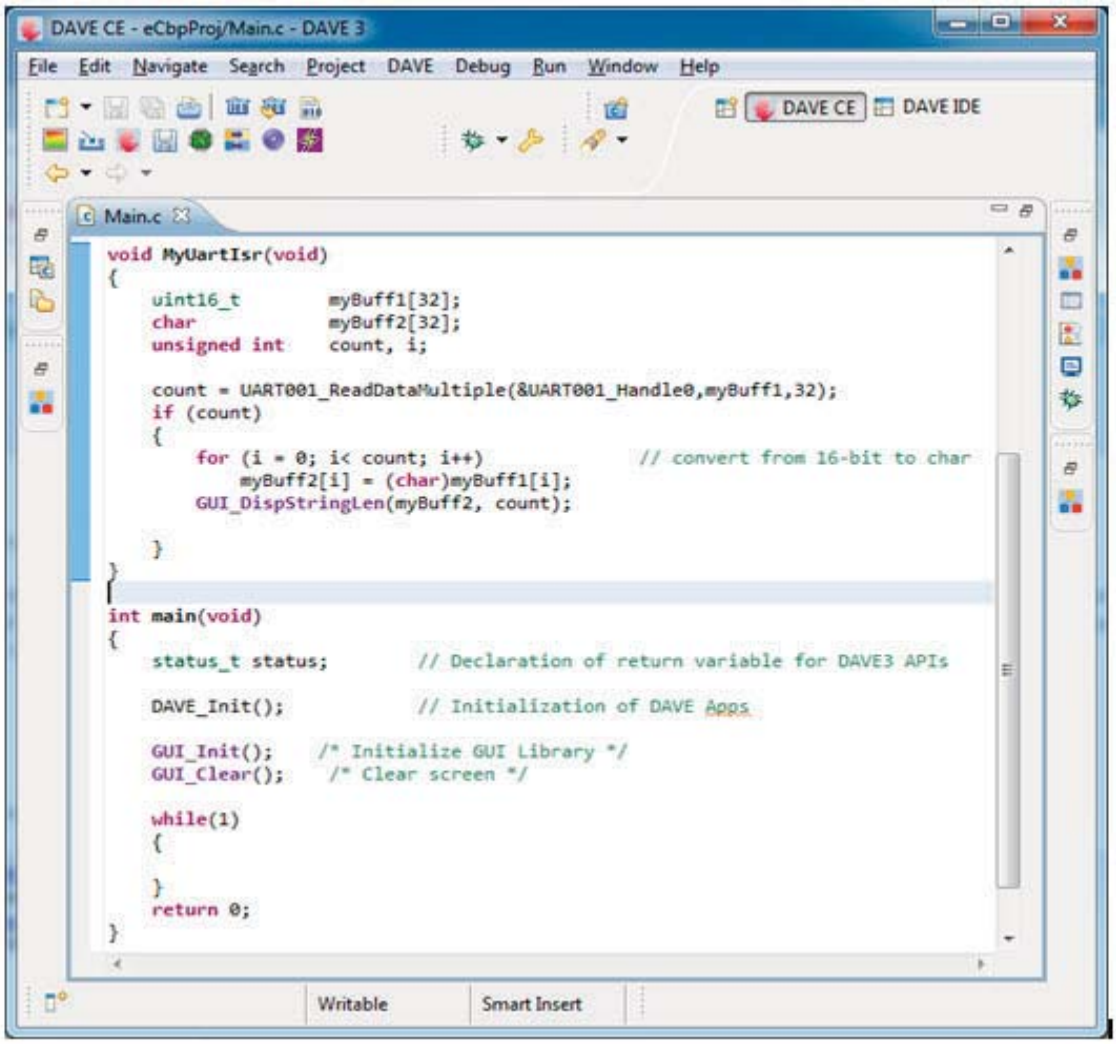

Figure 10: The ISR MyUartIsr (and the calls to GUI_Init() and GUI_Clear()) is the only manually generated code. Everything else is handled by the DAVE™ CE.

This function reads data from the FIFO until either the FIFO is empty or a certain number of bytes (in this case 32) are read. Also notice that the function is past an input parameter "UART001_Handle0". This is a key point about DAVE™ Apps. Since the Solver is free to assign the UART App to any of the available serial channels (provided it meets all of the users constraints), DAVE™ assigns each DAVE™ App a unique handle (id). This allows the code to be written independently of the actual physical peripheral that is used. It also allows multiple UART DAVE™ App instances to share the same API functions. So if your application requires six UARTs, you can simply add the UART DAVE™ App six times into your project but you will only see one instance of the generated .c and .h files.

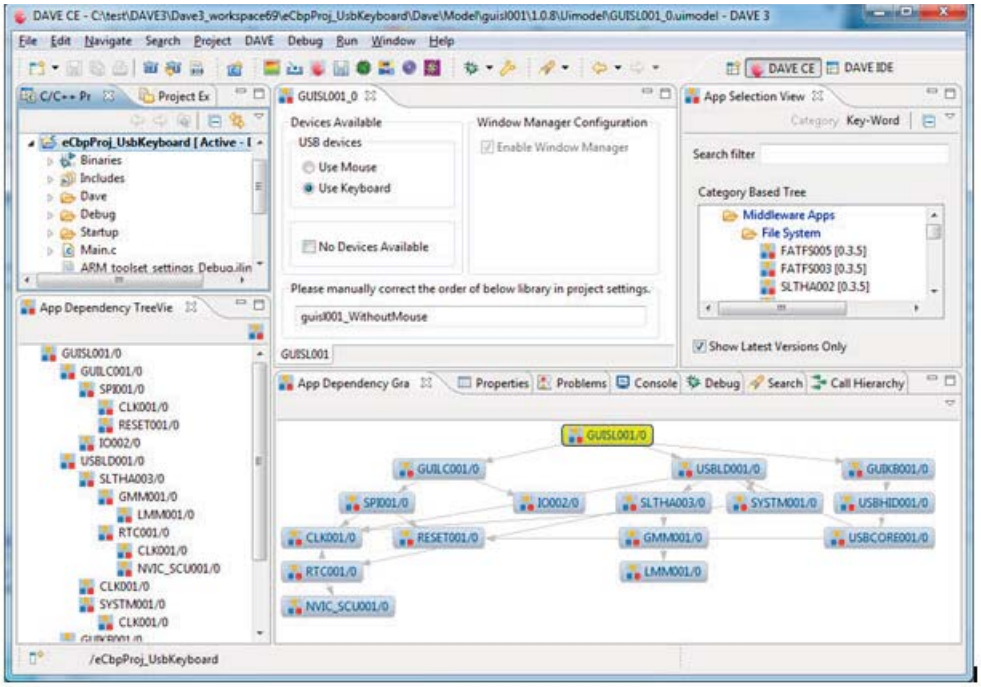

Figure 11: A complex system for a USB Keyboard with GUI is easily created using DAVE™

The previous example was very simple. It only took a couple of minutes from start to finish. The main complexity was in the OLED GUI libraries, and this was handled by the DAVE™ Apps. DAVE™ Apps are also available for much more complicated systems. For example, if we decide to dump the UART interface and connect a USB keyboard directly to the MCU (via the USB OTG interface on the XMC4500), we can just delete the UART & NVIC DAVE™ Apps from the project and select the "Use Keyboard" option from the Segger GUI DAVE™ App (GUISL001) UI. This will automatically insert the USB Keyboard DAVE™ App (USBKBD001) along with the USB driver (USBLD001 & USBCORE001) and USB HID (GUIKB001) DAVE™ Apps, and any other helper DAVE™ Apps that are required, as shown in Figure 11.

Unless a DAVE™ App uses a pre-compiled library, all of the source code is included in the project (in the DAVE\Generated folder) and can be easily viewed, modified or exported to other tool chains. Even the templates that show how the UI settings affect the generated code are provided in .jet script files for easy modification.

Summary

CBP as a software design methodology has been gaining in popularity. DAVE™ 3 now enables embedded programmers to benefit from CBP without sacrificing real-time behavior. DAVE™ 3 also eliminates the trade-off between reusability and use of MCU hardware resources by using virtual resources and the Solver.

DAVE™ 3 is available for free download from www.infineon.com/dave and supports all XMC family MCUs.

Currently all DAVE™ Apps are also available for free download. Within the next few months Infineon will release a SDK to allow individuals to create their own DAVE™ Apps to share or even sell via an online DAVE™ App store. Infineon currently releases new DAVE™ Apps every two or three weeks.

Further examples and training material are available at www.infineon.com/dave-support.

References

1 http://www.monkeys-at-keyboards.com/java3/1.shtml
2http://en.wikibooks.org/wiki/Computer_Programming/Component_based_software_development
3 Component-Oriented Programming Languages: Why, What, and How