

Standard API Would Significantly Accelerate Embedded System Development

Imagine being able to focus 80% of your efforts on improving the efficiency and feature set of your end product. Unfortunately, today's embedded system designers spend as little as 50% of their development effort on applications code. Instead, they are focused on the hardware (20%) and on making their applications software work on their hardware (30%). The bottom line is that time is wasted making software work on the target hardware. As a result, developers often have to choose between adopting new processor technologies to keep pace with the competition and developing new software to enhance features. It is typical to find that 40% of the feature set doesn't make it into the final product.

A major reason why so much effort is devoted to making the software work on the target hardware is that there is no standard API that allows software development to be independent of the hardware. The availability of a hardware independent API has enabled PC applications to flourish, but this capability has not been transferred to embedded applications. Windows CE and I20 are attempts to provide a solution, but neither has the flexibility nor capability for real-time critical embedded design.

In creating a development environment that automatically synthesizes tailored device driver, boot code and glue, Aisys has had to define an API for a series of different embedded processors. Based on this experience, it is now possible to propose an API that will allow development of software that is independent of the hardware implementation.

WHAT IS THE API?

There are many different levels of interface in a software system. The API Aisys has developed is the programming interface to the target hardware. It comprises device drivers, boot and low level glue code. There are two parts to the interface -- the interface between the RTOS and the hardware --and the interface between the applications code and the hardware.

Aisys has developed an API that provides an RTOS independent wrapper around the hardware. Further, its approach will allow you to extend the API to meet your own particular needs for peripherals that aren't covered anywhere else.

WHY IS THE API IMPORTANT?

The API standardizes the interface between the software and the hardware. A well-designed API will give you a series of important benefits:

1. Isolates the hardware from the software. This means

that the applications software can be hardware independent, greatly reducing the cost and risk of migrating to new hardware platforms.

2. Allows the driver code to be re-used, leading to greater efficiency and performance.
3. Allows the applications writers to access services easily by reducing ambiguity in the usage of the hardware.
4. Enables you to constrain the functions you need in the peripherals and so eliminate unnecessary "hardware features" that make the interface more difficult.

Unfortunately, there is often very little thought put into the API. It is handed down from other projects or determined arbitrarily from code downloaded from a website or included as part of an RTOS package.

DEFINING THE API - THE BUILDING BLOCK APPROACH

The approach Aisys has taken in developing the API is to treat each peripheral as a building block with the API set up to control the operation of these blocks. This concept is extended to include a set of configuration options that are actually part of the building block but can also alter its operation. Finally, the concept of building blocks is extended to include "pseudo peripherals" -- peripherals that don't exist but are a collection of items that can be treated as a peripheral (for example, clock set up and internal baud rate generators).

Building Block Basics

Each building block has three types of attributes:

1. **Services** it provides to the applications code that uses it. These services are C functions that are called by the building block user. They include some common functions that exist in each block and some specific functions that characterize each

FUNCTION NAME	SERVICE/FUNCTION PERFORMED
<code>void XXXInit(DeviceID)</code>	Initializes the DeviceID to a mode that was defined by the user interface. It is called once.
<code>void XXXConfig(DeviceID)</code>	Used to configure the device according to a run-time set configuration. This function is a superset of the Init function.
<code>int XXXTest(DeviceID)</code>	Used to test the device driver in the hardware environment. It is a simple test intended to be used mainly by the hardware designers while bringing up the board. It could also serve as a skeleton for a diagnostic-testing suite.
<code>void XXXEnable(DeviceID)</code>	Used to enable the device. By default, the devices are initialized at power up and initialization, and the enable function has to be called to start the driver operation.
<code>void XXXDisable(DeviceID)</code>	Used to disable the device. Every time the settings are changed, the driver should be first disabled, then the change can occur, followed by an enable call.

Table 1. Common Services

building block. Common functions include initialization function, configuration function and test function.

2. **Events**, which are usually interrupts intercepted by the hardware. Once received, the building block contains a placeholder for the event handling.
3. **Data items** that store the state of the building block. These data items contain values that the building block uses for its operation. Examples include baud rate, clock value, etc.

For each building block, these attributes are:

Common Services. All the device drivers have identical basic functionality common to all the drivers. This functionality includes the services/functions as stated in table 1.

Return Values. There are two distinct approaches to return values in embedded software design. One approach is to minimize return values and use **void** wherever possible. This approach minimizes memory and stack utilization. In low end, 8-bit applications, this approach is recommended. The other approach is to use return code as much as possible. Even when a function does not need to return a value, a success indication is returned to the caller, so validity checking and error handling become much easier.

For example, when using the function

```
void TxBuffer (Buffer_p);
```

the caller can't know if the transmission was successful or not. The function might have encountered problems in memory handling, or actual hardware setting (e.g. overrun), and the transmission did not occur. On the other hand, using the function

```
Int TxBuffer (Buffer_p)
```

where the return value could be the actual number of

bytes transmitted, or just a success/fail indication, gives the caller a much better insight into the progress of the program, and enables more robust software.

In high-end applications, we implement return values wherever possible.

Error Handling and Validation. Error handling is one of the toughest issues of software development. The main sources of errors are:

1. Coding errors (bugs)
2. Hardware errors
3. Communication statistical errors
4. Normal operation errors (no memory, queue full, etc.)

Good programming practice anticipates errors, and provides mechanisms to handle them without crashing the system. The overhead to take care of all errors is tremendous, and taking a purist approach will cause the code to inflate, execution time to increase and development schedules to grow exponentially. Defining the fine line between being practical and being careless is difficult. The building blocks defined in this document provide the following error handling for coding errors:

Passing Validation. How much validation should be done on the parameters passed to a function? Should each passed parameter be validated before the function is executed? The trade-off between a perfectly protected function and no protection is clearly in execution time and code size.

The function can contain validation code in the form of **assert** commands on each passed parameter. Since the function knows the expected range of parameters, it can contain a simple if statement that makes sure the parameters are within range.

All these tests can be enclosed within **#ifdef** brackets, so when in debugging mode, the function will go through the extra validation, and once the program reaches a high quality level, the **define** can be

removed, so the validation code will not compile and link into the application.

For example:

```
#define MAX_A 7
int func (a)
{

#ifdef debug
if (a<MAX_A) assert ("function func parameter a is above MAX-A")
#endif

return (2*a);
}
```

Success Indication.

Each function should return a value. If possible, the value should include a success/fail indication in its range (it could be a specific number, like 0 or -1, out of a range of valid numbers). Some functions can't fail (e.g. a function that shifts a value left). In such cases, there is no need to provide success/fail indication.

When calling a function, its success/fail indication should always be checked. Never assume that the functions are running successfully.

Hardware Errors Handling. Hardware errors could include parity error on memory, bus errors that cause a system level interrupt, watchdog timer expiration -- anything that tells the system that something went wrong. Whenever possible, these errors should be intercepted and handled as exceptions by the software.

Communication Statistical Errors Handling. These errors have specific protocol handling procedure, or are han-

dled according to the application specific inputs. Each building block has to intercept each and every such error and either process it internally, if appropriate, or pass it as an event to the application.

Normal Operation Errors Handling. The code should detect such errors (no memory, queue full) wherever possible, and report it to the caller. The handling of the error should be performed by the code that initiated the process.

Naming Convention and Code Style

Naming the functions is critical and is the key to the API. The functions must be portable and they must allow maximum reuse of the low-level code. For example, when there are two identical communication channels on a chip, it is desirable to have one driver

AD LOGIC

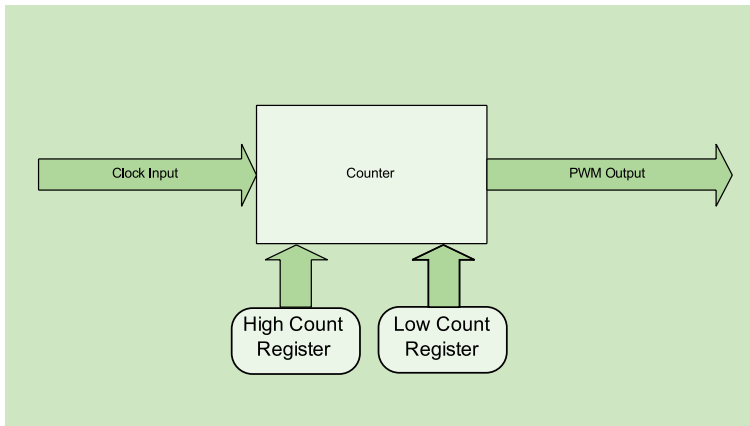


Figure 1. This block diagram shows a pulse width modulator, irrespective of how it is implemented.

control both of them, if possible. On the other hand, for increased readability, it is important to provide names to specific devices, reducing the potential for errors and improving the readability and maintainability of the code.

We have introduced a service-naming convention that is flexible enough for users to determine the names according to their specific preferences. Each function has an XXX prefix, which is replaced by a user-defined identifier. For example, XXXInit can be replaced with TimerInit, thus providing a general name for all the timers on the systems to be initialized; or, it could be called SystemClock1Init, which is very specific, and most probably will not be portable to other timers in the system.

To identify a peripheral in a group, the services contain a peripheral ID. The peripheral identifier is called DeviceID. For example there are four SCCs on the Motorola MPC860, so selecting one SCC to transmit a buffer would be achieved by:

```
SccsWrite(SCC1, Buffer.);
```

PWM THEORETICAL EXAMPLE

The pulse width modulator (PWM) building block is useful to generate an asymmetrical waveform on a timer's output pin. It is commonly used in devices that need a waveform with a given duty cycle, for example step motors.

Operation Sequence

The following attributes are used to define a PWM:

- Period - the number of counts of the clock that will represent a single pulse.
- Pulse width - the num-

ber of counts during which the pulse is at a high position. The pulse width must be shorter or equal to the period.

- High count - exactly as the pulse width.
- Low count - the period minus the high count. This is the number of counts for which the pulse is at a 'low' position.
- Clock Source- the clock that drives the counter. In most cases there is only one clock; in some cases it can be selected out of a list of possible clocks. This is an identifier that represents the clock that is going to feed the PWM counter.

It is assumed that the clock itself resides outside the PWM module, and is set somewhere else to a certain frequency.

There are two ways to operate the PWM system. One is by setting the HighCount and LowCount values, and this way, as they are changed the PWM value is changing. The other way is to use a fixed period, and then send SetPulseWidth commands to the system, which will set the Width of the pulse, but will keep the Period at a fixed length.

The Operation sequence starts with a call to

Pwmlnit (ID);

PWM initialization can set up all the initial values for the PWM module, including clock sources, initial Pulse Width and Pulse Period, etc., so the next step would be to enable the Building Block with a call to:

PwmEnable (ID);

This call will start the operation of the PWM with the initialization parameters. Then, every time the modulation parameters need to be changed, there are two options, either to call

PwmSetPulseWidth (ID, Width);

or

PwmSetPulseIntervals (ID, HighCount, LowCount);

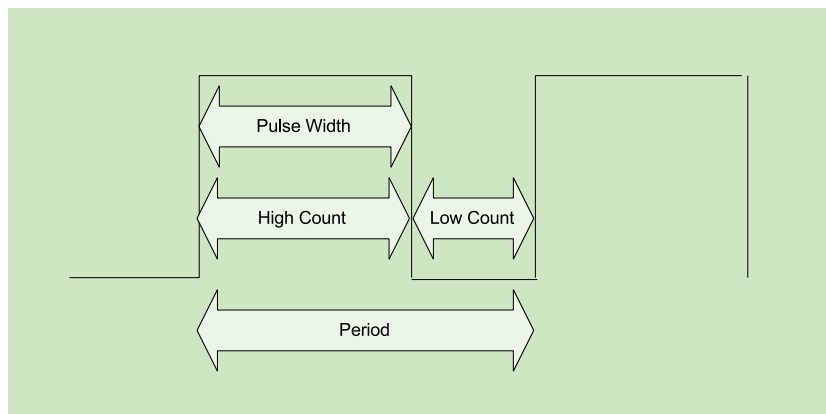


Figure 2. This diagram shows the output from a PWM unit.

If the modulation should stop, a call to

PwmDisable (ID, TRUE);

will halt the modulation operation and set the output to high. Restarting of the PWM operation can be done by using the PwmEnable service.

Services/Functions

void PwmInit(int Aid);

Initializes the PWM building block to its initial state. This function includes setting of the initial values for the modulator (the initial intervals), so the module will be able to start generating a series of pulses as soon as the module is enabled.

Calling Sequence:

- Aid - The identifier of the peripheral on which the building block is implemented.

Return Value:

void

If there is a timer peripheral in the system that is called Timer1, then the initialization function will call Pwm1Init (Timer1). The user must have the identifier defined prior to using this function.

void xxxSetClkSrc (int Aid, int ClkSrc);

Sets the clock source for this timer. If the timer has several possible clock sources, this function is used to select the one used for the specific implementation. In cases where there is no option, the clock source is set in the initialization function, and there is no need to use this function.

Calling Sequence:

- Aid - The identifier of the peripheral on which the building block is implemented.
- ClkSrc - An identifier for the specific clock source that will drive the PWM module.

Return Value:

void

int xxxSetPulseIntervals(int Aid, int HighCount, int LowCount);

Set the High and Low counts of the PWM waveform. The count is in clock ticks, and not in time units. The user can calculate the time if needed, or go back from time to clock ticks by dividing the time by the clock tick time interval.

Calling Sequence:

- Aid - The identifier of the peripheral on which the building block is implemented.
- HighCount - The value to which the timer will count when the output pin is at 'high'. After this number of ticks, the output of the PWM module will be switched to 'low', until the next 'high' value. The units here are input clock ticks. It is up to the user to convert between time or percents to clock ticks when using this function.
- LowCount - The value to which the timer will count when the output pin is at 'low'. After this number of ticks, the output of the PWM module will be

switched to 'high', until the next 'low' value. The units here are input clock ticks. It is up to the user to convert between time or percents to clock ticks when using this function.

Return Value: Success Indication.

0 - Success

1 - Combined count is bigger than the maximum clock count

int xxxSetPeriod(int Aid, int Period);

Set the Pulse period. This value is relative to the frequency in which the PWM works.

Calling Sequence:

- Aid - The identifier of the peripheral on which the building block is implemented.
- Period - The length of the period of the PWM module. This is the combined count of both the 'high' and 'low' intervals of the pulse. The units here are in clock source ticks. It is up to the user to convert between time to clock ticks when using this function.

Return Value: Success Indication.

0 - Success

1 - Period is bigger than the maximum clock count

void xxxSetPulseWidth(int Aid, int PulseWidth);

Sets the Pulse Width. This function is the actual modulation function. It should be called whenever the Pulse Width has to change, to signify a change in the data to be modulated.

Calling Sequence:

- Aid - The identifier of the peripheral on which the building block is implemented.
- PulseWidth- The value to which the timer will count when the output pin is at 'high'. After this number of ticks, the output of the PWM module will be switched to 'low', until the next 'high' value. The units here are input clock ticks. It is up to the user to convert between time or percents to clock ticks when using this function.

Return Value: Success Indication.

0 - Success

1 - PulseWidth is bigger than Period

void xxxEnable(int Aid);

Starts the PWM signal output. Until enabled, the PWM output is stable, and no modulation occurs.

Calling Sequence:

- Aid - The identifier of the peripheral on which the building block is implemented.

Return Value:

void

void PwmDisable(int Aid, boolean StopLevel);

Stops the PWM operation and sets the output at the requested value (HIGH will set it to high, LOW will set it to low).

Calling Sequence:

- Aid - The identifier of the peripheral on which the building block is implemented.

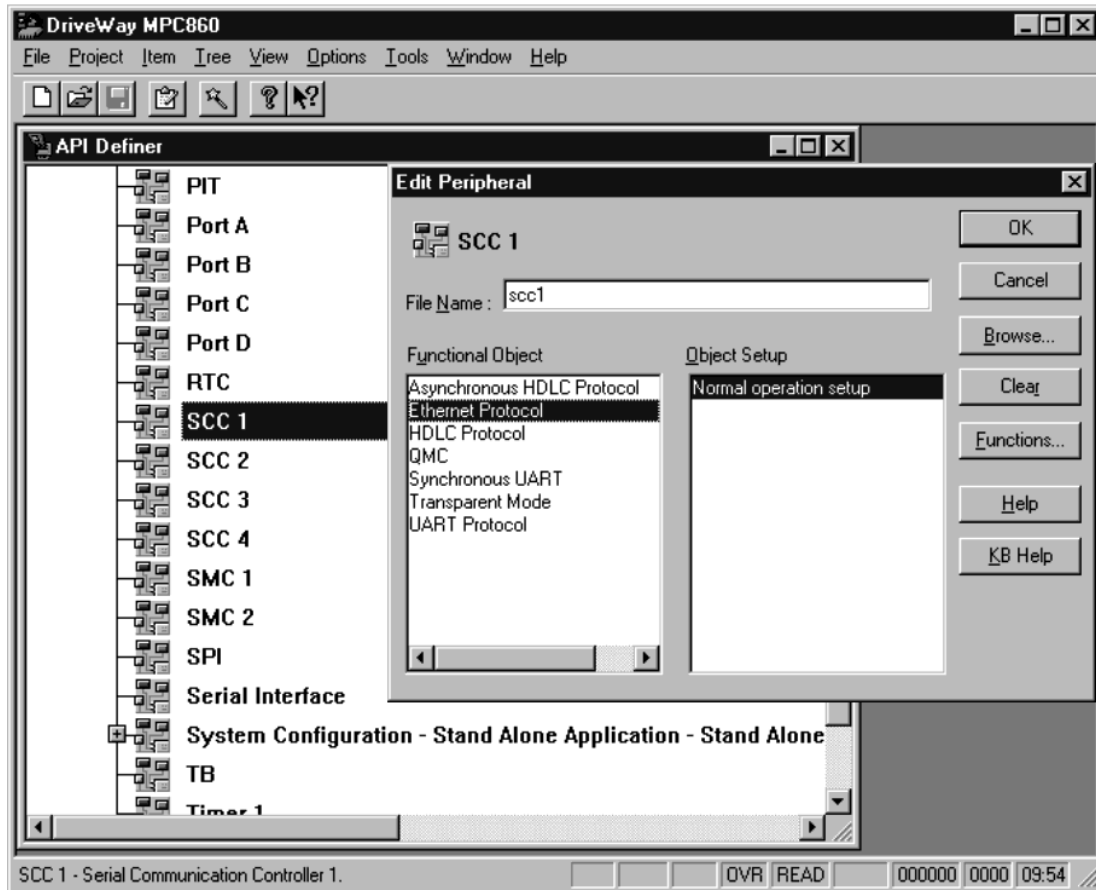


Figure 3. Configure the peripheral

- StopLevel - When the PWM stops its operation, it is required to set the output level at a requested level. Otherwise, if the function will be called during a 'high' interval, the output will remain in a 'high' state, and vice versa. This parameter directs the PWM module to set itself to a requested level.

Return Value:

void

If the function is called while the PWM is already inactive, the level of the PWM output should still be set to the StopLevel parameter.

Data

The following data item should be defined to enable proper usage of the PWM building blocks:

yyyMAX_COUNT - The maximum count for the timer used to implement the PWM module. This will be used to make sure that the values set for the time intervals do not overflow the MAX_COUNT value.

ETHERNET PRACTICAL EXAMPLE

In this example, DriveWay will be used to step through building a device driver to support Ethernet. This is an example of using DriveWay-MPC860 to configure the peripherals for the MPC860T microprocessor from Motorola. This will serve as an example of how the API has been applied to an automation system and allows you to rapidly build code that conforms to a consistent interface.

Configure the peripheral

Most peripherals are multi-function and so the initial step is to select the peripheral and determine the function you want it to perform. Figure 3 shows how a multi-function peripheral is constrained through DriveWay's user interface. All the functions the peripheral will support are presented and the user determines which functional mode is required.



Figure 4. Select the services that the peripheral will supply

AD NATIONAL INSTRUMENTS

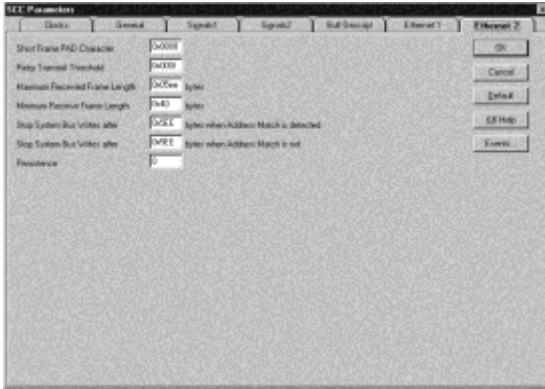


Figure 5. The data screen

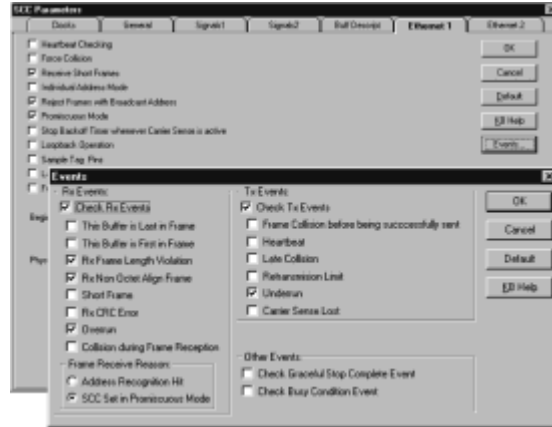


Figure 6. The event screen

Select the services

Having selected the function of the peripheral, DriveWay allows you to select which drivers you need. Each of the drivers in the set conforms to the API standard. Figure 4 shows how the user can select the services that the peripheral will supply. This assumes that the function has already been selected (as shown on the dialog box).

Tailor the functionality

The last step is to fully tailor the operation of the peripheral. DriveWay does this by providing you with a set of data and event screens that you can tailor. Some of the screens are the same irrespective of the function of the peripheral (e.g. clock sources), others are function dependent. The example below deals with Ethernet specific data and event screens.

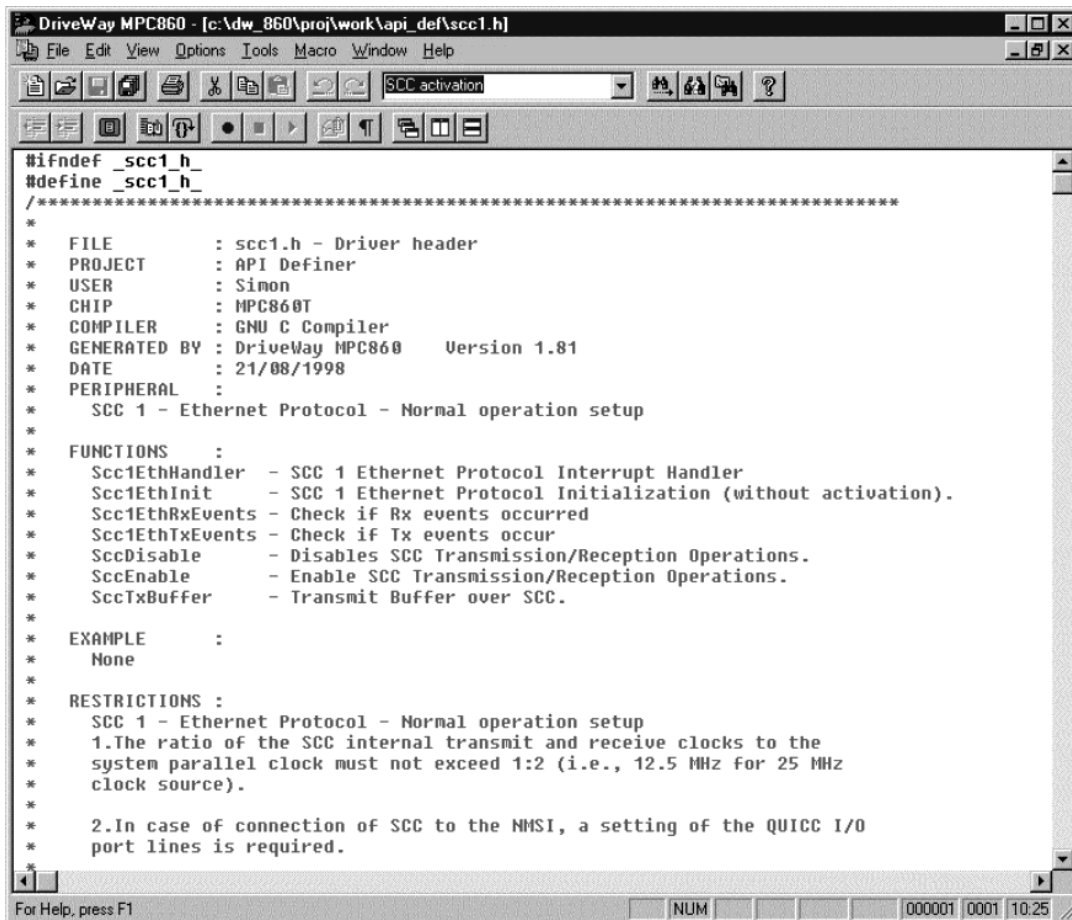


Figure 7. scc1.h

```

DriveWay MPC860 - [C:\DW_860\PROJ\WORK\api_def\scc1.c]
File Edit View Options Tools Macro Window Help
SCC activation
/*****
 * FUNCTION   : Scc1HandleEthRxEvents
 * Description :
 *   Handle Scc1 Ethernet Rx Events
 *
 *****/
void Scc1HandleEthRxEvents( BD_P Prd )
{
    /*
     * Handle Ethernet Rx Frame Length Violation Event.
     * A frame length greater than the maximum defined for this channel was
     * recognized (only the maximum-allowed number of bytes is written
     * to the data buffer).
     */
    if ( RxBdFrameLengthViolation( Prd ) )
    {
        /* Insert your code between the DWS USER CODE comments! */
        /* DWS_USER_CODE( TAG_Scc1HandleRxEvents_4 ) */

        /* DWS_END_USER_CODE() */
    }

    /*
     * Handle Rx Nonoctet Aligned Frame
     * A frame that contained a number of bits not divisible by 8 was
     * received, and the CRC check that occurred at the preceding byte
     * boundary generated an error.
     */
    if ( RxBdNonoctetAlignedFrame( Prd ) )
    {
        /* Insert your code between the DWS USER CODE comments! */
        /* DWS_USER_CODE( TAG_Scc1HandleRxEvents_5 ) */

        /* DWS_END_USER_CODE() */
    }
}

```

Figure 8. The Receive events handler

Data

The data screen for Ethernet allows you to alter the values for certain conditions, as shown below.

Events

The event screens allow you to select which Ethernet events you want to trap. Below, it is possible to see how DriveWay generates code that allows you to determine what to do when these specific events occur.

GENERATED CODE FRAGMENTS

The first code fragment is scc1.h. This shows the header which summarizes the project details and outlines the functions which make up the Ethernet drivers.

The second code fragment shows the Receive events handler. This corresponds to the event selections that were made in the event panel of the final configuration. DriveWay uses "magic markers" that allow you to add your own code. In this case, the code is set up to detect frame length violations and then you can add your own code to determine what action should be taken.

CONCLUSION

The device drivers, boot and glue code that interface the applications code and the hardware are too often written without enough thought being put into the critical role this code plays in the system. In particular, the

value of applying a consistent API gets lost in the rush of completing the project on time.

Using a consistent API will reduce ambiguities and thus time wasted in understanding how to use the driver code. It will enhance productivity by enabling the applications code writers to have a consistent interface to the hardware. It will enable greater efficiency in the driver code both in terms of performance and memory footprint.

In the future, a consistent API will reduce the time and risk to port to new hardware and will enable greater efficiency by allowing teams to share drivers and reuse code more effectively.

Finally, a consistent API allows for automation tools to expedite the process, enabling you to spend more time adding value to your end product. ■

Shaul Gal-Oz is founder, president and CEO of Aisys, Ltd. He previously managed a team of 40 engineers (in Israel) developing systems in the fields of signal processing and artificial intelligence. In addition, Mr. Gal-Oz also spent three years in the US managing a \$30 million classified project with RCA Government Communication Systems (Moorestown, NJ). He received his bachelors degree in Electronic Engineering from Ben-Gurion University in 1981. He can be contacted at shaul_gal-oz@aisysinc.com.