

trace (Interrupt) can be displayed, for which reason this type of low-cost trace is mainly found with Cortex-M based microcontrollers. The maximum possible speed of the SWD interface is independent from the core clock used and virtually only limited by the basic technology. The integrated parity check of the telegram guarantees a reliable transmission. An additional acknowledge field (ACK) contained in the SWD protocol ensures correct execution of the operation.

The new vendor-independent IEEE 1149.7 (compact JTAG or cJTAG) standard was ultimately developed to meet current and future demands of board and system tests. IEEE 1149.7 offers additional new features to support power management, application debug and device programming while maintaining full backward compatibility with the existing IEEE1149.1. A total of 6 classes are defined, which can be implemented by IEEE 1149.7 compatible TAP controllers (TAP.7). Class 0 defines the adherence to 1149.1 compatibility,

Class 1 enables additional command protocol for reset and power management, Class 2 offers the capability of BYPASS for chip internal TAPs for optimization of the shift chain, Class 3 defines the capability of star coupling several TAP.7 controllers, Class 4 provides new scan protocols on the basis of only two signal lines, and Class 5 allows the use of up to two application-specific data lines e.g. for real-time trace. Classes 4 and 5 are especially interesting for embedded applications where cost and performance are particularly important. On the one hand here also, as with both of the other solution approaches presented, only two pins are needed for the communication. On the other hand, the data throughput is considerably increased by various optimized scan protocols. The IEEE1149.7 standard is still new and corresponding implementations are rare. According to this writer's understanding, a complete synthesizable IP core for cJTAG is so far only available from IPextreme. Other various committees including NEXUS and MIPI, which are

also involved with debug interfaces, will certainly take the cJTAG standard into account. Meanwhile, most microcontroller users agree that the almost twenty-year old JTAG standard is only suitable to a limited extent to meet today's demands for speed, pin count and robustness. However, the question remains; what is the ideal future solution? As the described examples show, different manufacturers and interest groups are following similar fundamental principles in the search for an answer, but there is certainly no real standardization. Therefore, the challenge for debug and test tools consists of support for all new interfaces. A prerequisite for such universally usable and at the same time commercially acceptable solutions is certainly, in addition to a modular adapter concept for different connectors (figure 4), also a flexible tool architecture. With special FPGA equipped debug devices, such as pls universal access device 2 (UAD2), it is already today possible to implement nearly all debug protocols without much additional effort. ■

# Automotive ECU software development with SCADE Suite

By Masaru Kurihara, Electronics Engineering Department, Fuji Heavy Industries

*In this article the model-based development environment SCADE Suite is introduced to the existing process for development of a motor control unit in an electric vehicle, with particular reference to safe multi-thread executions.*

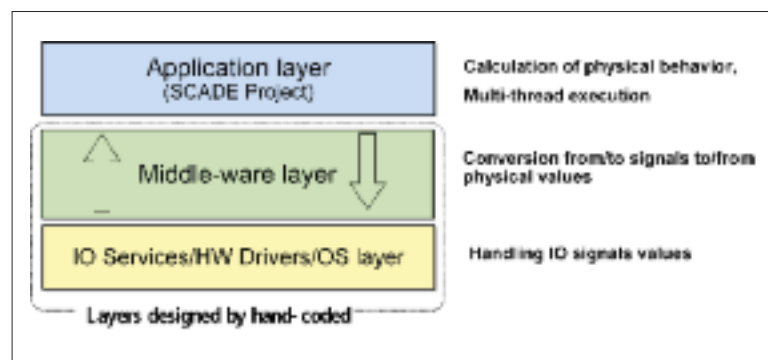


Figure 1. Static software architecture design

■ As safety requirements for vehicles and drivers advance, and new markets demand new functionality, the size and complexity of automotive embedded software increases correspondingly. New concepts like hybrid vehicles and electric vehicles create further needs, driven by environmental protection and emissions legislation. The software development

time for electric vehicles is longer than for conventional vehicles because their systems are designed from scratch, while those of conventional vehicles advance by incremental functional developments. Consequently it is difficult to assure a certain level of quality and reliability with the traditional methodology employed in the existing process, especially given the

limited time-to-market. As a methodological solution, we first introduce model-based development to make the existing process consistent and seamless, from software requirements down to code integration, in terms of reducing development time, avoiding introduction of human errors and improving development efficiency. To achieve these objectives in a

model-based development process, the following principles should be observed: 1) automate interface definition between SCADE Suite generated application code and lower layer code (middle-ware, hardware drivers, operating system), 2) ensure consistency in data handling among different threads of execution, 3) cover the process from software requirements down to code integration including verification activities by a single tool, and 4) minimize verification effort according to modification and change of requirements or specifications. In the following, we present two examples of automating the interface definition between SCADE application code and lower layer code and ensuring consistency in data handling among different thread executions in details.

Following these principles, we introduce SCADE Suite into the existing development process for the development project of a motor control unit managing a motor in an electric vehicle. To enable a smooth start with model-based design, software architecture is firstly redesigned according to a layer structure consisting of three layers as shown in figure 1. At the top level, the application layer, which is independent of targets and operating systems, is dedicated to design application behaviour under multi-thread execution. In the middle of

the structure, middle-ware converts all physical values from the application layer to signals and all signals from hardware drivers and the operating system layer to physical values. The application layer is fully developed by SCADE tools. In the development process, the behaviour of the application is described by SCADE Editor, with concurrent verification activities by simulator and model test coverage achieving requirement-based tests, code generated automatically by automatic code generator while the traceability from the requirements to tests and code is entirely managed by the requirements management gateway. Thanks to the SCADE certified software factory concept, the last two principles mentioned above are easily observed.

Since the middle-ware layer is developed in the traditional way, based on manual coding, human errors may be introduced into the interface when SCADE generated code is integrated into the middle-ware. Taking advantage of SCADE tool openness, execution of customized scripts in the SCADE tool environment enables the automatic creation of the interface between SCADE application code and middle-ware code in order to avoid the human errors introduced at the integration stage. Thus, the first principle is also observed.

What has not been considered yet with SCADE tool deployment is ensuring consistency in data handling among different thread executions. Since embedding implementation of safe interaction across different thread executions in code is error-prone even for expert programmers, as the complexity of the application is higher, safe architecture at SCADE model level should be clearly defined. Once it is fully defined, it can be automatically implemented in the code generated by the SCADE KCG code generator. Thanks to this modular concept in SCADE language, whatever SCADE model is described within the defined architecture, the whole application remains safe for different thread executions.

Default data types corresponding to SCADE basic types are generated at code generation from SCADE models. Since the default types can be replaced with user types at code generation, the types to be replaced should be identical with the ones defined in middle-ware codes. At the early stage of development, not only the application specification, but also the hardware and middle-ware specifications may be changed frequently. If the interface between middle-ware and application code is maintained by users, one modification may impact both layers, thus inconsistent data type errors may be introduced by users. In order to have robust management of data types for interface development, extracting the data types from middle-ware code and creating the type definition file to be provided to SCADE are automated by Tcl script execution. Figure 2 shows the example of extraction from middle-ware code and creation of type definition file for SCADE. SCADE generates `kcg_int` for signed integer, `kcg_real` for float (or double) and `kcg_bool` for unsigned char as default while the middle-ware defines float as user type `float32`, signed long int as `sint32`. With Tcl script execution, both types are compared and found identical. As a result, the type definition file is created.

Once identical types are defined, data structures are next extracted. The extracted data structures are automatically converted to the SCADE language notation and imported into the SCADE project to make them available anywhere in the SCADE model. This automated creation of type definition can be applied to complex data structures. As soon as data type definition in middle-ware code is changed, execution of the Tcl scripts updates the type definition for SCADE application models.

The motor control unit needs to manage various functions. However due to limitations on target performance, the software architecture should be optimized to meet performance requirements. To achieve the objective, functions

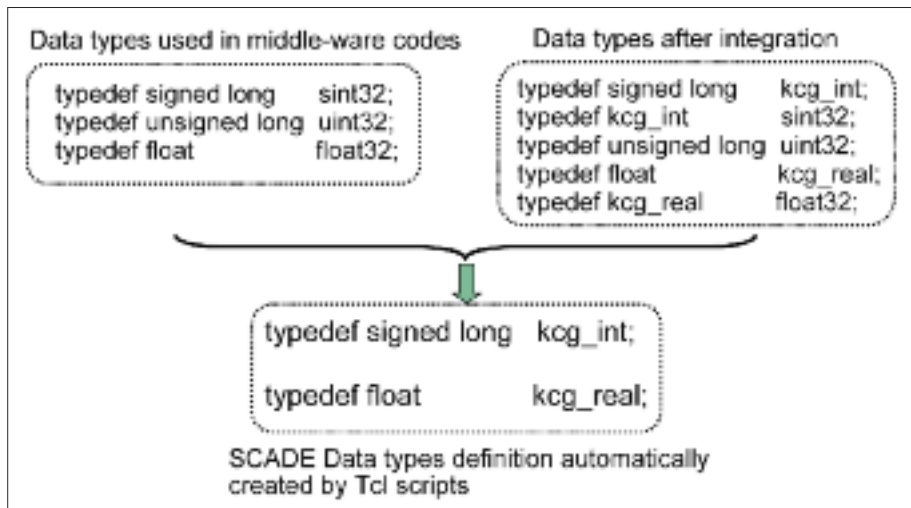


Figure 2. SCADE Data type definition automatically created from middleware code

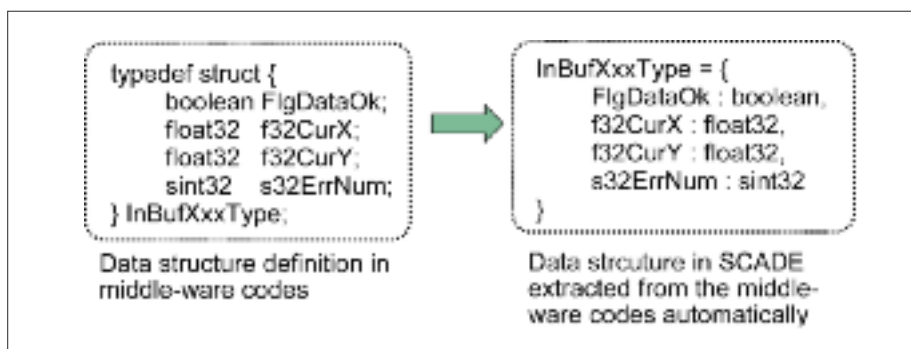


Figure 3. Data structure definition in SCADE based on data structure in middleware code

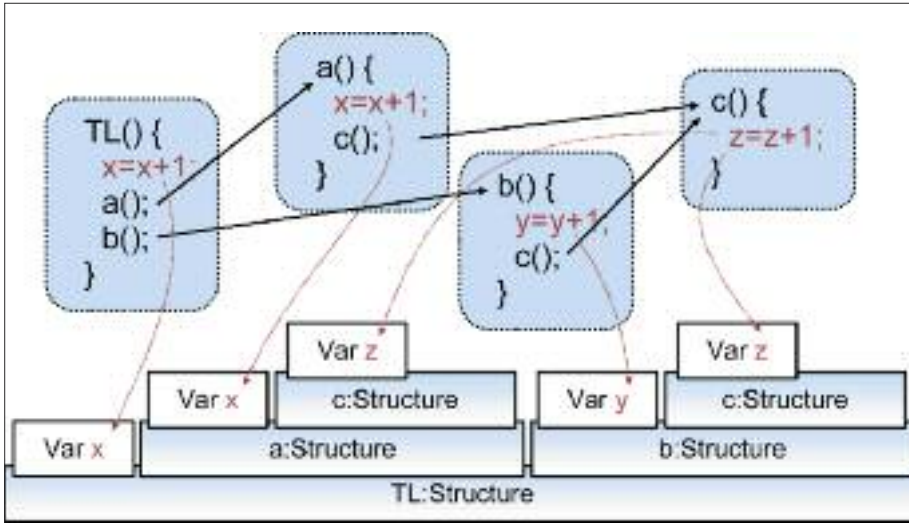


Figure 4. Function and its unique data storage area

are normally executed by appropriate order in a process, that is, multi-thread execution. In a motor control unit, four different threads are designed at the top node level of the application layer, which is fully designed by SCADE models. With SCADE language constructs such as Boolean activation, activate if or activate when, the top node can be easily designed. But an interesting issue is to see how safe the code generated from SCADE models designed by the specific constructs is. From the point of view of safe codes, the following two key rules should

be observed in SCADE model structure: prevent variables from being overwritten by other thread executions, and ensure consistent access to structured variables among different thread executions. In multi-thread execution, a thread execution is normally suspended by interruption of other thread executions while it is being executed. When the function is suspended, what happens to variables in the function should be examined at code level for safe execution. In the case that variables in the function need to store only values, but not their states, they should be

declared as auto variables on the stack allocated to the function. On the other hand, in the case that the variables need to store not only values, but also their states, unique data storage area per instance of the function should be allocated.

Figure 4 explains the idea. In a top level function TL(), a value is assigned to a variable x and two functions named a() and b() are executed by turns. In the function a(), a value is assigned to a variable x and function c() is executed. Both TL() and a() have same variable x, but due to the unique data storage concept, they do not come into conflict with access to the value of x. This applies to the relation of function a() and b() in execution of the function c(). Both functions have their own access to the value of a variable z in the function c() from function a() and b(). Keeping to the rule that a single access path reaching to the value of a variable exists per function prevents variables from being overwritten by other thread executions. Based on the safe concept, let us see how SCADE code generator can tackle this issue.

The answer looks simple. It generates the code meeting the requirement stated above as default. For the example shown in figure 4, a set of function arguments and state variables corresponding to a node is declared as a context based on a modular concept. The context for a

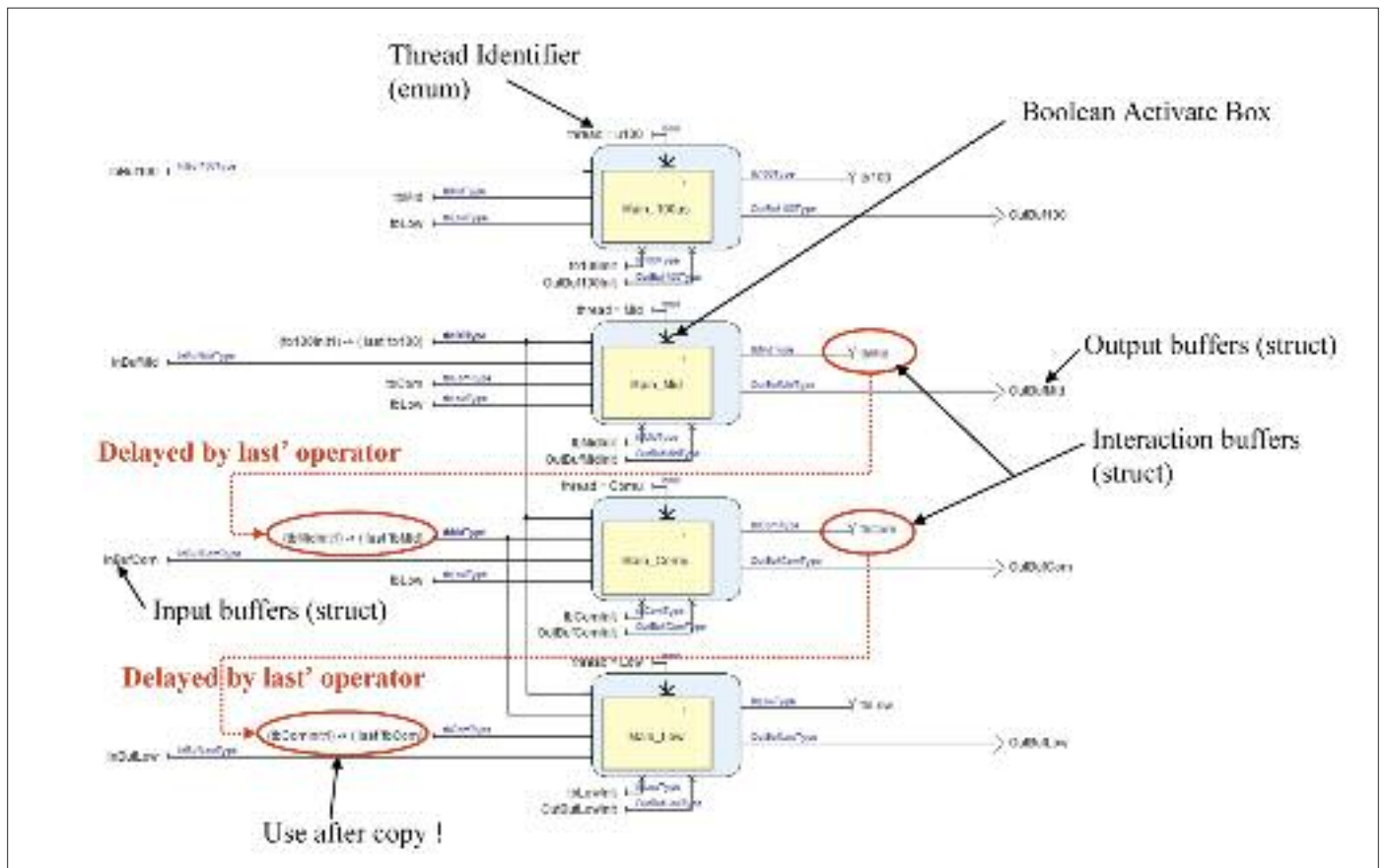


Figure 5. Safe SCADE top node architecture design

node which contains other nodes has its own arguments and state variables if used, and its calling nodes context. Thus, the code structure enables safe reuse of the functions outside any time.

In the last section, how to prevent variables from being overwritten by multi-thread executions has been explained. But this is not enough to design a complex application under multi-thread executions. What about the case that data is produced and read by different thread executions? For example, there are four different threads connected to each other to execute required functions in the motor control unit application layer. In this architecture, the critical problem is inconsistent data access among different thread executions. There are two significantly unsafe cases. One of these is to read the output variables of a function by other functions executed with different threads while update of the values is not completed. The other is to update the output values of a function while other functions executed by different threads are still reading the values. That could be a cause of non-deterministic behaviour of the SCADE application under multi-thread executions. In order to ensure safe data access among different thread executions, SCADE model structure at the top level satisfying the following four cases in reading and writing data should be defined as architecture.

First, a node with low-priority thread execution reads values of output variables which are produced by a node with high-priority thread execution. Execute memory copy prohibiting interruption, then read values from the copied variables. In SCADE generated code, execution of memory copy method for structure data utilizes memcpy() function from the generic li-

brary as default. Since the memcpy() function can be replaced with any other copy function by users, user-defined memory copy function with no interruption is given at code generation. In the SCADE language, an equivalent construct to memory copy function is last operator. It is defined to access the last value of the variable. Usage of last operator to access to the output variables produced by a node with high-priority thread execution on SCADE model considers memory copy in code level.

Second, a node with low-priority thread execution writes values to output variables which are referred by a node with high-priority thread execution and in case of structure data, write values with no interruption after all members become available. In SCADE generated codes the structure data is written after all members become available as default. So, at SCADE model level, no specific construct is required. Third, a node with high-priority thread execution reads values of output variables which are produced by a node with low-priority thread execution and refers values of the variables directly. Fourth, a node with high-priority thread execution writes values to output variables which are referred by a node with low-priority thread execution and writes values to the variables as normal way.

Considering these four cases, a top node architecture model is designed for four different thread executions as shown in figure 5. Four SCADE nodes are designed to execute by a Boolean activation operator at top level. One of four enumerated conditions which are thread definition is always matched to activate the corresponding SCADE node in the thread. The last operator is used at several places in the model to read values of copied output variables pro-

duced by a node with high-priority thread execution. In the model, a node named as Main\_100us is given the highest priority according to software requirements, while lower priorities are given to the remaining nodes Main\_Mid, Main\_Comu and Main\_Low by turns. In the case that the node Main\_Mid needs to use values of outputs named as tb100 produced by Main\_100us, last operator is used to read the values after they are copied. On the other hand, when the node Main\_100us reads the values of outputs named as tbMid produced by Main\_Mid, last operator is not used, but directly referenced. In summary, the SCADE model architecture enabling multi-thread executions is clearly designed by combination of Boolean activation and last operator, and can be a template for applications requiring safe multi-thread executions.

Now that the SCADE top-level node is defined, the remaining task is to complete the detailed design in the top nodes allocated to threads. Before going into detailed design, the top node architecture model with multi-thread executions should be integrated into the middle-ware layer. This validation is required once for the first design of the top node model as long as middle-ware and lower layers are unchanged, because the validation is aimed at integration test of application code and middle-ware code. Instead of detailed design in top nodes, stab is enough for integration test here. Once stab is ready in top nodes, SCADE Simulator is used to see if the application top node behaves as required. Then, proceed to code generation of the top node with stab and pass the all codes generated to another team working for integration. At this stage, application engineers can concurrently design detailed behaviour with SCADE for each thread execution. ■

## Embedded News

Free E-mail Newsletter for Europe`s Embedded Engineers

- Chips & Components
- Tools & Software
- Boards & Modules



[www.embedded-control-europe.com/newsletter](http://www.embedded-control-europe.com/newsletter)