

Implementation of Software Timers in Model-Based Design for Body Control Software Applications

Jinming Yang, Jason Bauman and Al Beydoun

Electrical Systems Division, Lear Corporation

Copyright © 2009 SAE International

ABSTRACT

Selecting a proper implementation method for software timers in automotive body control applications is one of the crucial steps in the process of software development. We found it particularly true when model-based design (MBD) methodology and automatic code generation techniques are utilized. From our past experience and lessons learned, we have noticed that many software defects are primarily due to inappropriate implementation of software timers.

Generally, in model-based design, three options are available for implementing software timers in the application layer of a standard embedded software architecture: (a) the timers are implemented using temporal logic functions furnished in modeling toolsets (e.g., the “*after*” or “*before*” functions in Stateflow® of MATLAB® tools), (b) timer functions are hand-coded as custom C functions that are called from feature models in which timing is independent of the calling rates, and (c) custom counters are directly employed to count external timing events.

In reality, technical challenges manifest themselves in later development phases when pieces of auto-code generated from feature models are integrated and all timers are physically driven by system clocks inherited from hardware timers of a microcontroller. Moreover, it is usually the case that time-out parameters for timers are required to be configurable and the calling rate of a functional model containing software timers needs to be adjusted during software integration. This will even add more complexity to timer designs. In this work, the MathWorks’ toolset including Simulink®/Stateflow® and

Real-Time Workshop Embedded Coder® was used. The methodology introduced in this paper has been successfully applied to production projects.

INTRODUCTION

Programmable and/or configurable temporal functions are widely used in automotive body control applications where finite state machines are a necessary means to implement various operational modes and to keep track of feature execution. In general, a timeout could indicate that input signals and/or conditions have stabilized so that a triggered event or action can then be allowed to happen (for instance, software debouncers [1]). Also, a timeout could signal that some functions and associated transitional actions involved in a feature model have misbehaved and a corrective action (for instance, a “Retry” or “Abort” action) is required [2]. In body control applications, software timers are ubiquitous, both in the Application Layer and quite often in the Service Layer from the viewpoint of the standard architecture [3] for software on ECUs.

From our past experience and lessons learned, we found that a great number of software bugs associated with state machines are due to inappropriate timer designs and incorrect control flows around software timers. The problems fall into the following categories:

- (1) A timer fails to be reset and/or reloaded when it is prematurely cancelled. This may result in wrong timeout timing when the timer is re-activated for the second time and thereafter.

- (2) A timer fails to be stopped and/or terminated and keeps running when it is forced to transition out of a state by a higher-priority condition. This may cause timer counters to overflow.
- (3) Due to certain design pitfalls, a timer could be kept being reset repeatedly so that timer expiration never occurs or timeout is not controllable.
- (4) Timer resolution or timeout range is incorrectly selected so that either timeout occurs beyond the required range of accuracy or counters for the timer overflow before it expires.

The selection of ways to implement timers can be a quite application-specific process. When model-based design and automatic code generation processes are employed in software development, the following criteria should be considered to select a proper way for implementation in order to make robust designs:

- (a) In model-based design, timers and timing functions should be fully testable in both simulation and an integrated code environment.
- (b) Expiration of timers should be solely contingent upon their timing parameters and shall be independent of the calling rate designated by the task scheduler.
- (c) When the number of software timers involved in a system is considered as “many” (typically, when it is from *tens* to *hundreds*), the way of implementing timers should promote a commonized platform for managing timer counters/indices, timer validation, and memory configurations.
- (d) In model-based design, timer functions can be either hand-coded or automatically code-generated. In this case, timers should be designed to facilitate the interface between hand code and auto-code.
- (e) Timers should be designed in such a way that expiration timings and tolerance are configurable in order to test the timers and also meet new or changed requirements.

The above-mentioned criteria may not be applicable to a given implementation approach and a given application at the same time. Rather, some of them may be critical to certain timer implementations and certain applications, while others may be indispensable measures in other applications.

In this paper, three options for implementing software timers are presented and the characteristics, advantages and disadvantages of each option are analyzed. We shall also examine generated code for timers from a C code viewpoint for different methods of timer design. We then demonstrate our ways to cope with the challenges in timer design. In addition, techniques for interfacing feature models with manually

written source code for timers and timer validation using simulations are also introduced. Some design issues and typical design pitfalls will also be summarized.

TYPICAL WAYS OF IMPLEMENTING TIMERS

To describe typical ways of implementing software timers with a simple example, we shall design two timers to implement a pulse generator, using a Stateflow® chart, in which the “ON” and “OFF” time are both configurable.

METHOD#1: USING TEMPORAL LOGIC FUNCTIONS

A 20% duty-cycle pulse train generator can be easily implemented by using the Stateflow chart in Fig. 1, where two timers are implemented using temporal logic functions facilitated by modeling tools (e.g., the “after” or “before” functions in Stateflow® or MATLAB® tools). Note that the “tick” is chosen to be at a rate of 10 milliseconds in the current design. By running a simulation for the chart in Fig. 1, the pulse train generated is illustrated in Fig. 2. We use “T1_Cfg” and “T2_Cfg” to denote the timeout parameters.

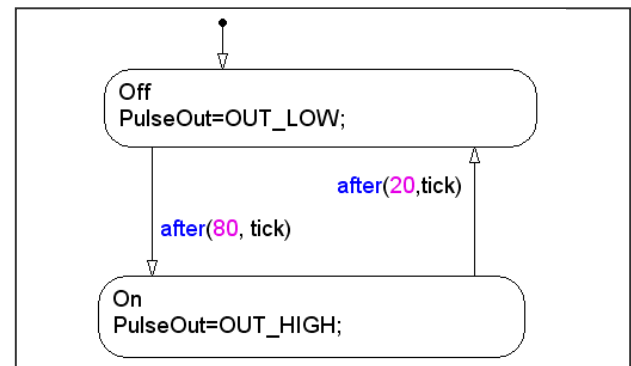


Fig. 1: Two timers using the “after” function are utilized to implement a pulse train generator which produces a 20% duty-cycle pulse train.

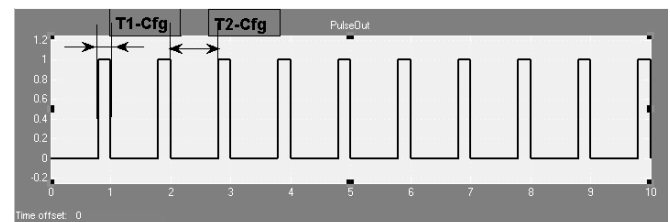


Fig. 2: Output from the Stateflow chart above. This pulse train generator will be implemented using different kinds of timer implementation below.

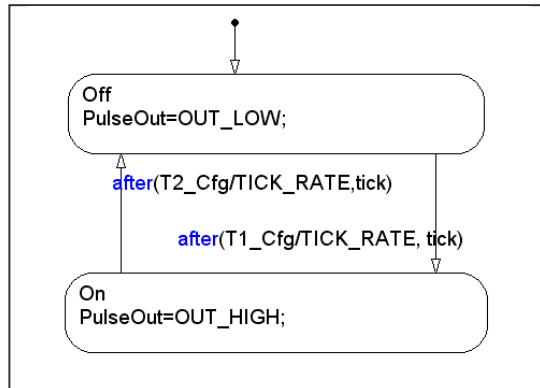


Fig. 3: The timers are implemented using the “after” function but the timing parameters and calling rate are employed in the “after” function.

Note that in Fig. 1, numbers are directly used and timer parameters are determined by the calling rate. This is normally not desirable from a software design perspective, since various software modules may need to be designated to different time slots in the scheduler with different task rates. Improvement can be achieved by using symbols representing the timeout parameters and calling rate. The symbols are then made available at its system level for configurations. This is shown in Fig. 3. This improvement has the benefit that even if the calling rate is changed, the timeout time will remain the same and therefore the timers can still work as required.

METHOD #2: HAND-CODED TIMER FUNCTIONS

In Method #1, the timings for timers are dependent on the rate of the software module being called, which is determined by the scheduler of the operating system. If all the application software modules in a system run at one or a few simple rates, and the system scheduler is well established and consolidated, the first method can be a very convenient way to design software timers, and consequently issues of code generation and software integration can easily be resolved. However, software modules are often required to be executed at different rates due to functional requirements and/or constraints from CPU utilization and worst-case execution timing. Specifically, certain software modules need to be executed more frequently than others, and there is usually a need to alleviate the CPU load for the worst cases by evenly distributing software tasks into different time slots designated in the scheduler. To this end, the scheduler that dispatches tasks at different rates could always be subject to changes and tunings during the software integration phase. Obviously, the calling rate and execution orders that are determined by the scheduler are crucial to system functionality. Thus, there is a strong reason why timings for software timers should not rely on the calling rates built in the scheduler. Therefore, the method using temporal logic functions may not always be applicable.

As far as software running on a physical microcontroller is concerned, the timing base for software timers is inherited from its hardware timers, which are configured and controlled by low-level drivers residing in the driver layer of the software hierarchy. It is always desirable that such a timing base can be utilized as a single source for all the software timers. However, such a timing base from the driver layer is not readily employed by software in the Application Layer which in our case, consists of automatically generated code from feature models.

A feasible solution is to hand code the timer functions which are driven by the timing base and then are called by the automatically generated software from the Application Layer. Conceptually, then, the timer functions run in a separate thread so that the timings are independent of the application software and its scheduler. The application models can still be developed and tested separately while this is going on. The downside to this method is that the interface between auto-code and hand code inevitably adds some complexity, which will be described in more detail later.

When this method of timer implementation is selected, the model for the pulse generator example will look like what is shown in Fig. 4. Note that two C functions, “StartTimer()” and “CheckTimer()” are directly coded in the model. These two functions have to be hand-written and included in the “Simulation Target” in the SimuLink® model. More details about implementing these timer functions shall be given later in this paper.

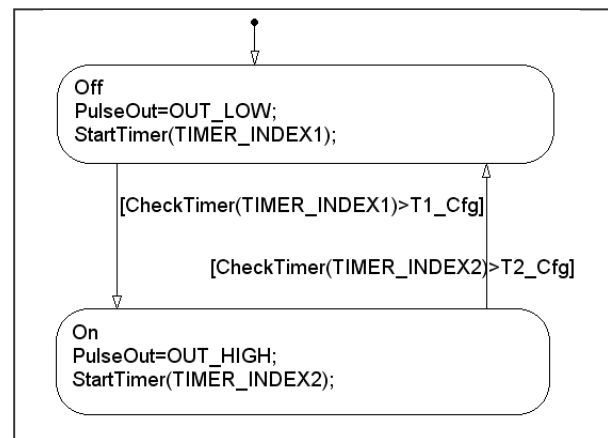


Fig. 4: The timer functions are hand coded and timers are implemented by calling these functions from inside the model.