Considerations for Safety Critical, Real-Time Embedded Software Development

By Matthew Noonan, Project Manager, Resource Group's Embedded Systems & Solutions division

Compiler

Arguably the most important choice in any safety critical embedded software development. The choice of compiler will affect what integrity level you can achieve and will identify families of microprocessors / microcontrollers that can be used. It can also impact what real-time operating system is to be used.

Target

The target is the other candidate for the most important choice to be made. There are enormous possibilities in today’s market for embedded target choices. These need to be refined according to a large wish list:

Compiler Compatibility – can a compiler be sourced that can approve the code on the chosen target to the appropriate safety integrity level? Is the associated RTOS compatible/certifiable?

Speed – from the project estimates, can the target perform to the necessary speed?

Ability – Does the target need particular functions in terms of I/O or communications? Does the target have the necessary ports to do this? Has it got any in-build signal processing/communication handlers?

Memory – Does the target have sufficient program memory allocation? Is there sufficient RAM? Is an amount of Flash or NVM available? Is the size acceptable for the project estimates?

Package – Is the target provided in a package that is acceptable to the hardware providers? Hardware in safety-critical applications also has higher levels of certification to meet and some packages are very hard to verify. This needs to be checked with hardware providers before any choice is finalised.

Availability – One consideration of safety critical development is the protection against obsolescence. The chosen target needs to be available for the life of the product. Can the chosen target be sourced from many suppliers? Is it an older chipset and so likely to be replaced within a number of years?

Real-Time Operating System (RTOS)

Choices need to be made early within a project about how the software scheduling will be handled. Will a bespoke operating system or scheduler be built and certified as part of the development or will a commercially available operating system be used? Most RTOS are tied to core architectures (e.g. PowerPC architectures are now mostly VxWorks) for each family within a chipset so this needs to be considered as part of the target choice. RTOS certifiability is also a consideration that needs to be added to the factors in compiler choice.

For many systems the choice of a bespoke scheduling program will be taken. This provides a lot of control and benefits but also greatly adds to the complexity of a system. The first task in designing a system of this type is to define what type of scheduling is necessary; the less complex the scheduler, the easier to verify the function:

Fixed/Static Scheduling – The schedule is calculated at or prior to compile time to produce a pre-defined schedule at run-time. This can be the simplest form of scheduling but will only work for very deterministic systems. A system with a lot of interrupts or with a large range of iteration rates may not fit within the available time-scales with a fixed schedule. This is also an inefficient use of the processor time and means that the processor speed may need to be higher to compensate.

Dynamic Scheduling – This brings the scheduler into the realm of a real-time operating system. Dynamic scheduling
employs an element of the code itself acting as the operating system to decide, during run time which tasks should be scheduled in what order. Here many different disciplines need to be considered:

Priores – Allocation of priorities to particular schedule items is necessary in most dynamic scheduling, even if a clear priority between the tasks is not obvious. Priorities can take many forms:

Shortest time remaining – Processes with the least time to their process deadline are given the highest priority. This can work in a safety critical environment under certain circumstances but verification activities will need to prove that no combination of process orders can cause a time-slot overrun, a large analysis of Worst Case Execution Time is necessary.

Fixed priority – each process has a fixed ranking priority assigned at compile-time. This ensures that the most critical systems will be dealt with within their deadlines. This can work in a safety environment but only in either a small system or one with a large amount of allowable Jitter.

First-In-First-Out – The simplest form of dynamic scheduling in principle. It does what it says on the tin; the first process ready for execution is run first, etc. The safety critical considerations are similar to Shortest Time Remaining.

Round-Robin – Very simple and technically no allocated priorities. The process involves allocating equal pre-determined time slots to each process. There are many efficiency benefits to this but it is not practical for safety critical systems as the suspension of processes at the end of the time slots makes the verification of correct data-flow a near impossible task.

Multi-level – Processes are separated into groups of priorities. It is an amalgamation of First-In-First-Out and Shortest Time Remaining.

This can work well on safety critical systems. As with others, an analysis of Worst Case Execution Time will be needed.

Pre-Emptive – This is another consideration on the type of scheduling. Where processes have distinct priorities, whether the arrival of a higher priority process will suspend the current process or not (whether it will pre-empt or not) is a difficult question for safety critical systems. The benefit of pre-emptive scheduling techniques is a more efficient use of processor time and more importantly faster response to what could be dangerous situations being detected. The downside is that to have processes suspended means that the dataflow between processes is no longer entirely deterministic as the point of suspension cannot be predicted. There are methods to mitigate this using software architecture but they are not simple or easy to prove.

Whichever scheduling algorithm is used there are some common considerations that need to be analysed. In normal real-time software development, these considerations would give a measure of how robust or effective the product was; in safety-critical systems, these play a higher role of giving evidence that the system is robust and can be certified:

Worst Case Execution Time – Whether for static or dynamic scheduling, the worst case execution time for each code function needs to be considered to allow the scheduling calculations to show that a time-slot overrun and subsequent real-time unhandled exception cannot occur. There are many methods of determining the worst case execution time for each code function. In principle, this value needs to represent the time taken to execute, in the target environment, the longest path through the code function. The scheduling calculations must then show that in a worst case scheduling scenario (with all processes ready for execution, including the maximum possible number of interrupts), that the sum of the worst case execution times can still be handled by the processor without any overruns.
Jitter – This is the allowable amount of tolerance on a code function’s iteration rate. For example, if the function is required to be executed every 50ms but has an allowable jitter of 5ms, the scheduler has more flexibility in where it is scheduled (45 to 55ms from when it was last executed).

Interrupts – Interrupts in safety-critical applications need to be very carefully planned. Due to the need to show that no combinations of the schedule can cause an overrun in time-allocation, the sum of the worst case execution times of all interrupts in a system need to be added to the worst case execution time of each code function to show that this cannot overrun the window allocated. As such, interrupts are normally kept to the absolute minimum and then only for very short processes.

Considerations and Practices

There are some considerations that are applied to some real-time embedded software systems that become more important in a safety-critical environment:

Execution Time Delta – The amount of change in the execution time of any individual code function should be limited when employed in a safety critical function. I.e. Any number of branches available should be of the same length. This allows much more accurate calculation of scheduling needs and makes analysis of the deviation in this much more deterministic.

Architecture – The architecture of a software system can have very large impact on the ability and cost of verification. The use of global data should be avoided if at all possible and the use of passed parameters is much more encouraged. Clear regulation about areas of software function and responsibility also help to provide evidence that each consideration of the requirements has been dealt with.

Requirements – The level, accuracy and quality of the software requirements can reflect enormously on the final cost of the project as well as the ability to accurately estimate completion dates. A period of time spent adjusting and critiquing requirements can pay dividends in the long run. Also, the work flow adherence, as with the rest of safety-critical development must be adhered to very strictly. With requirements there is also the difficulty in accurately tracing the split between software implementation and functional requirement to handle; as the saying goes “Writing software to meet requirements and walking on water; both are only possible when they are frozen”.