JavaScript is disabled in your browser. Please enable JavaScript to proceed. You can’t perform that action at this time. $define1Generating Comments As described in Section State Machines, QM provides extensive support for modern Hierarchical State Machines (HSMs)↑ (UML Statecharts). From the code engineering point of view, state machines are the most "constructive" element of the UML and the support of state machine code generation is the most valuable aspect of QM. This section describes the state machine implementation strategies and coding aspects for hierarchical state machines in C and C++. Class ToastOven with a hierarchical state machine used in the following examples of code generation As described in Section State Machine Base Classes, QM supports two state machine implementation strategies, depending on the selected base class for the application-level state machine: State Machine Constructor Apart from selecting the superclass (base class) in the Class Property Sheet, the constructor of the application-level state machine must call the appropriate base class constructor. For example, a state machine class derived from QHsm must call the QHsm constructor and class derived from QMActive must call QMActive constructor. State Machine Constructor in C The Section Class Constructors in C describes how to model class constructors in C. has been described in the B State Machine Constructor in C++ Action Code in C Accessing Attributes Accessing Event Parameters Action Code in C++ Accessing Attributes Accessing Event Parameters $define1Generating Comments StateSmith is a cross platform, free/open source tool for generating state machines in multiple programming languages. The generated code is human readable, has zero dependencies and is suitable for use with tiny bare metal microcontrollers, video games, apps, web, computers... It avoids dynamic memory allocations for the safety or performance inclined. The above is my current plan, but I'll gladly help anyone add a new language. I'm hoping contributors will help me with this effort. It is tricky though... The fundamentals-1 webpage has simple interactive examples that let you explore most StateSmith features. Want to jump right in and just try it!? The below tutorials use new StateSmith features that are more user friendly. They use different diagram tools, but mirror each other fairly closely otherwise. If you are new to state machines, then prepare to level up your toolbox! They are incredibly helpful for certain applications. Why StateSmith? I couldn't find a quality state machine code generator that met my needs, had an attractive license, and was enjoyable to use. Before I created StateSmith, it was always a pain trying to manually synchronize a hand written state machine with a drawing. Urgent client requests come in and you update the code, but do you and your team always remember to update the drawing? Probably not and so the rot begins. Documentation trust issues arise and as designs get larger, the effort to ensure the diagram is accurate starts to become quite punishing. Now that we use StateSmith at my work, I never have to worry about the above. I love generating fully working code from the documentation. Incredibly helpful for teams and communicating with clients. The StateSmith-examples repo has a growing list of examples showcasing different application uses. We use StateSmith in a fair number of production projects at my work. It's been super helpful. Other companies are using StateSmith in production as well (consumer electronics, autonomous vehicles, ...). StateSmith has a strong suite of tests (730+) and behavior specification coverage. The specification integration tests read a diagram file, generate executable state machine code, then compile and execute that code in another process and ensure that the state machine behavior is exactly what was expected. The same suite of integration tests run for each supported programming language. This strong test base gives me confidence. It also allows us to refactor and optimize StateSmith without fear of accidentally breaking specified behavior. The StateSmith GitHub wiki has a good amount of documentation right now, but always feel free to ask a question. YouTube channel: statesmith Join us on discord. Feel free to open a github issue. Or you can use the project's discussion space. Code generation tool written in Python for C++ hierarchical state machines. The basic idea is to design your state machine graphically in PlantUml and then use the PlantUml input file also as an input file for FloHsm.py to generate C++ code. PlantUml can draw states and transitions, but does not have knowledge of basic state machine concepts such as event, actions and guards. A state transition in PlantUml is simply written as State1 --> State2 : comment Here, 'comment' is a free format string that is printed along with the transition arrow in the state diagram. This is fine for a diagram, but in a concrete state machine implementation a state transition must be triggered by an event and it may or may not have an action attached to it. Also, depending on some guard condition, the transition may or may not be performed. FloHsm has solved this problem by defining additional state machine language elements that are ignored by PlantUml, but shown in the diagram as plain text. Here are some examples of valid FloHsm transitions. Basically everything before the semicolon is standard PlantUml syntax, everything after the semicolon is FloHsm syntax. Transition triggered by event E1 Transition triggerd by event E1, but only if boolean guard expression G1 evaluates to true Transition with action A1, with and without guard S1 --> S2 : E1 / A1 S1 --> S2 : E1 [G1] / A1 python FloHsm.py statemachine.txt See Source/Generated/TestCompositeState for an example. Open the .puml file in plantuml and have a look at the test for using the generated code in C++ For now, FloHsm does not parse the @startuml and @enduml keywords that are required by PlantUml. The solution is to write the state machine description in a text file sm.txt. This file is used for FloHsm. A second file sm.puml only contains the following lines and is used to render the diagram in PlantUml @startuml !include sm.txt @enduml After running FloHsm.py, most of your state machine code is generated, but there are two things that the tool cannot generate. The implementation of the actions and the guards. You need to write them yourself in your state machine class. They are however present in as pure virtual functions in the state machine base class from which your state machine derives, so you just need to override and implement them. You also need to initialize the state machine before use. In short Derive your state machine class from StateMachineBase Implement al pure virtual functions from StateMachineBase (the actions and guards) Make sure to call InitStateMachine() from your state machine class before using it, e.g. from the constructor or an init method Calling the event methods (inherited from StateMachineBase, you don't need to do anything here) on your state machine will now cause state transitions, execution of actions, etc... There are a couple of tests that demonstrate most of the FloHsm capabilities. Please find them in Source/Generated/Test*. There is a .puml file for viewing in PlantUml and a .txt file that is used for FloHsm. Generate the state machine files and run the tests. This is as easy as building the project in Visual Studio 2017 and running the test executable. The tests should be easy to compile on other platforms and compilers, but development and testing was only done on Windows with Visual Studio More documentation and implementation coming soon. See the issues page for identified open issues State behaviors can also have guard conditions. The transition from ON2 to ON_HOT has a guard condition [count >= 3]. This transition will only be taken on the INCREASE event if count >= 3. Transition guards can be any code that evaluate to a boolean result. In this example, the guard tests a state machine variable, but it could call a function, a bunch of functions, do some math... Another interesting thing in this example is that we specify the order of the ON2 behaviors for the INCREASE event. We want to run count++ before testing if we should transition with [count >= 3]. You can read more about state behaviors here. Auto clear diagram highlights. State machine variables count: 0 Sinelabore enables developers to effectively combine event-driven architecture, hierarchical state machines, model-based design and automatic code generation. A payback is usually given already immediately. SinelaboreRT focus is on generation of readable and maintainable code from flat or hierarchical UML state machine diagrams. With its unique features the tool covers perfectly the requirements of embedded real-time and low-power application developers coding in C / CPP. The generated code is independent of CPU and operating system. Many systems are likely candidates for implementation as finite state machines. A system that must sequence a series of actions or that must handle inputs differently depending on the mode it is in is often best implemented as a finite state machine. Typical examples are control-logic-oriented applications such as metering, monitoring, workflows and control applications. For IoT applications where parts of the application are implemented in Java / Python / C# / Lua / Rust / JavaScript / Go or Swift, the code can also be generated in these languages. The Sinelabore code generator runs on any OS that supports a modern Java Version e.g. Windows, Linux, macOS or from within a container. By generating code that can be compiled with virtually any compiler, and the ability to integrate with your existing IDE, build process or continuous integration system, the code generator can be quickly integrated into any project. Configuration is stored in a plain text file which allows customisation of generated code to exactly your needs. Generated code has production-quality. It is based on nested switch/case and if/then/else statements. It is easy to read, understand and debug if needed. The generated code requires no compiler specific tricks except standard language features. This means that if the worst comes to the worst, you can easily change or expand the code by hand. Can be used with any 8-, 16- or 32-bit CPUs. There is no run-time environment needed like with some other solutions. Fits well in different system designs. The code generator does not dictate how you design your system. Therefore it is no problem to use the generated code in the context of a real-time operating system (VxWorks, FreeRTOS, embOS, RTEMS, …) or within an interrupt service routine or in a foreground / background (super loop) system. There will be no problems when using static code analyzers. Generated cpp code passes clang-tidy and is cpp11 ready (modernize-*). Set configuration parameters accordingly. Avoid bugs that can waste countless hours of developer and end-user time before they are found. Developers spend a lot of their time coding state machines by hand. And have to do it whenever the design changes. Sinelabore avoids the error-prone and tedious hand-coding by generating high-quality source code directly from the state machine design document. No gap between design and code anymore. The documentation is always up to date. An integrated state diagram editor makes it easy to get started and allows you to create state diagrams within minutes. The entry barrier is significantly lower compared to full-fledged UML tools. A series of tutorials (see sidebar) explains step by step how to use the integrated diagram tool. Use the code generator only for those parts of your software that benefit from state machine modeling and code generation. Use your existing development environment for all the other code. The code-generator does not dictate an “all or nothing” approach as many other commercial tools. Automatic robustness tests, test-case generation, tracing and simulation Extensive Manual with getting started section The code generator runs locally on your developer workstations, build servers or continuous integration servers. It does not use an internet connection and will never collect nor submit data, code, statistics, analytics, or any other information from your system over any channel. To get an impression of the powerful capabilities of the tool download the demo version. Checkout the examples folder to see the generated code. Follow the “Getting Started” pages on this website. The manual contains a basic introduction into state-machines in case you need a refresh. Read the sections related to your UML tool and the language backend you want to use. If no UML tool is already in place take a look at the built in state machine diagram editor. To run the code you have two options. Run the examples on your PC. The example folder contains examples for all supported modelling tools and various languages (C, CPP, …). The examples realizes a microwave oven and can be executed and tested. Play with the model and enhance it. Regenerate the code and learn from the warning and error messages. Run examples on a Micro-Controller e.g. a MSP430 evaluation board using Energia. An example with all details is available on github. Sinelabore Code Generator is used worldwide by companies of all sizes, from well-known multinational organizations to smaller independent companies and consultants. The code generator is also used in a wide range of industries. “Sinelabore has helped me implement the behavior of a complex, asynchronous system. All the UML 2 elements I needed are available. I like that I don’t have to draw the state machine, then separately implement it and keep these two synchronized; this saves me time and reduces the potential of bugs. The error checking to make sure the state machine is valid is also useful. —- Daniel Bedrenko / Software Developer @ BPS…tec GmbH” “Thank you again for providing such great tool!” “… wir nutzen Ihr Produkt schon seit vielen Jahren und es hat sich als zuverlässiges und wertvolles Werkzeug erwiesen …” “We like Your Tool, infact we will give intro for another local company next week.” Study done by “Laboratory of Model Driven Engineering for Embedded Systems @ CEA in France” with the title “Complete Code Generation from UML State Machine” write in their report “ … without optimization, Sinelabore generates the smallest executable size ,,,”. Reactive systems are characterized by a continuous interaction with their environment. They typically continuously receive inputs (events) from their environment and − usually within quite a short delay − react on these inputs. Reactive systems can be very well described with the help of state machines. State machines allow to develop an application in an iterative way. States in the state diagram often correspond to states in the application. The resulting model helps to manage the complexity of the application and to discuss it with colleagues from other departments (and domains). Details can be added step by step during the development. Even during creation, the Code Generator can check the state diagrams for consistency (Model Check). Its logic can be simulated and tested. This ensures that the state machine behaves as intended. State machines are very useful for control-oriented applications where attributes such as reliability, code size, power consumption, and real-time behavior are particularly important. There are different ways how to integrate state machines in a specific system design. Some design principles are more applicable for developers of deeply embedded systems. Others more relevant for developers having not so tight resource constraints. The SinelaboreRT code generator supports you in the creation of the state based control logic. Generated code fits well in different system designs. The code generator does not dictate how you design your system. Therefore it is no problem to use the generated code in the context of a real-time operating system or within an interrupt service routine or in a foreground / background system. In this design an endless loop — typically the main function — calls one or more state machines after each other. It is still one of the most common ways of designing small embedded systems. The event information processed from the state machines might come from global or local variables fed from other code or IRQ handlers. The benefits of this design are no need for a runtime framework and only little RAM requirements. The consequences are: All housekeeping code has to be provided by the designer Main loop must be fast enough for the overall required response time In case of extensions the timing must be carefully rechecked again Example: void main(void){ … sm_A(); sm_B(); … } This design is like the one presented above. But the state machine receives its events from an event queue. The queue is filled from timer events, other state machines (cooperating machines) or interrupt handlers. Benefits: Events are not lost (queuing) Decoupling of event processing from event generation. Consequences: A minimal runtime framework is required: Timers and Queues Main loop must be fast enough for the overall required response time A minimal runtime framework for C is available here: It offers timers and queues. The intended usage is as follows: Each state machine has an own event queue Eventually a state machine requires one or more timers (single shot or cyclically). A state machine can create as many timers as needed. When creating a timer the event queue of the state machine and the timeout event has to be provided. For different timers it makes sense to provide different timeout events. To make the timer work, a tick counter variable has to be incremented cyclically from a timer interrupt (e.g., every 10 ms). The tick frequency should be selected based on the minimal required resolution of the timeout times. A tick() function must be called in the main loop to check if any timer has expired. In case a timeout has happened the provided event is stored in the event queue of the state machine. The main loop has to check if events are stored for a state machine in its queue. If there are new events they are pulled from the queue and the state machine is called with the event. Example code with two state machines shows the general principle: // tick irq void tick(void){ pulseCnt++; } void main(void){ …. // create two queues for two state machines and init the timer subsystem fifoInit(&fifo2VendingMachine, fifo2VendingMachineRawMem, 8); fifoInit(&fifo2ProductStoreMachine, fifo2ProductStoreMachineRawMem, 8); timerInit(); … while (1) { uint8_t evt; // // Check if there are new events for the state machine. If yes, // call state machine with event. // bool fifoEmpty = fifoIsEmpty(&fifo2VendingMachine); if (!fifoEmpty) { fifoGet(&fifo2VendingMachine, &evt); vending_machine(&vendingMachine, evt); } fifoEmpty = fifoIsEmpty(&fifo2ProductStoreMachine); if (!fifoEmpty) { fifoGet(&fifo2ProductStoreMachine, &evt); product_store_sm(&productStoreMachine, evt); } // any new timeouts? tick(); } As indicated in the figure above also other state machines or interrupt handlers might push events to the queue of a state machine. An example how to do this is shown below. // add event evErr to a state machine queue. void ISR_Btn1() { fifoPut(&fifo2VendingMachine, evErr); } In low power system designs a key design goal is to keep the processor as long as possible in low power mode and only wake it up if something needs to be processed. The design is very similar to the one described above. The main difference is that the main loop runs not all time but only in case an event has happened. The timer service for the small runtime framework is handled in the timer interrupt. A skeleton for the MSP430 looks as follows: void main(void){ // init system while(1) { // check event queues and run the state machine as shown above … __bis_SR_register(LPM3_bits + GIE); // Enter low power mode once __no_operation(); // no more events to process } } // Timer A0 interrupt service routine. If the timer // function tick() returns true there // is a timeout and we wakeup the main loop. #pragma vector=TIMER0_A0_VECTOR __interrupt void Timer_A0(void) { bool retVal=false; P1OUT |= BIT0; // toggle for debugging retVal = tick(); if(retVal){ // at least one timeout timer fired. // wake up main loop bic_SR_register_on_exit(LPM3_bits); } P1OUT &= ~BIT0; // toggle for debugging // no more events must be processed } The following temperature transmitter using a MSP430F1232 header board with just 256 bytes of RAM and 8K of program memory is based on this design principle. For more information on how to use state-machines in low-power embedded systems see here and here. Sometimes state dependent interrupt handling is required. Then it is useful to embed the state machine directly into the interrupt handler to save every us. Typical usage might be the pre-processing of characters received by a serial interface. Or state dependent filtering of an analog signal before further processing takes place. Using state machines in an interrupt handler can be useful in any system design. For code generation some considerations are necessary. Usually it is necessary to decorate interrupt handlers with compiler specific keywords or vector information , etc. Furthermore interrupt service handlers have no parameters and no return value. To meet these requirements the Sinelabore code generator offers the parameters StateMachineFunctionPrefixHeader, StateMachineFunctionPrefixCFile and HsmFunctionWithInstanceParameters. The example below shows an interrupt service routine with the compiler specific extensions as required by mspgcc // generated state machine code for an irq interrupt (INTERRUPT_VECTOR) IntServiceRoutine(void) { /* generated statemachine code goes here */ } To generate this code, set the key/value pairs in your configuration file the following way: StateMachineFunctionPrefixCFile=interrupt (INTERRUPT_VECTOR) HsmFunctionWithInstanceParameters=no If the prefix of the interrupt service routine requires to span more than one line the line break ’’ character can be inserted as shown below: StateMachineFunctionPrefixCFile=#pragma vector=UART0TX_VECTOR__interrupt void Prefixes for the header and the C file can be specified separately. In this design each state machine usually runs in the context of an own task. The principle design is shown in the following figure. Each task executes a state machine (often called active object) in an endless while loop. The tasks wait for new events to be processed from the state machine. In case no event is present the task is set in idle mode from the RTOS. In case one or more new events are available the RTOS wakes up the task. The used RTOS mechanism for event signaling can be different. But often a message queue is used. Events might be stored in the event queue from various sources. E.g. from within another task or from inside an interrupt service routine. This design can be realized with every real-time operating system. Only the event transport mechanisms might differ. Benefits: Efficient and well tested runtime environment provided from the real-time operating system Prioritization of tasks, scheduling available State machine processing times decoupled from each other. Consequences: Need of a real-time operating system (complexity, ram usage, cost …) In the how-to section an example of this pattern is presented with FreeRTOS. The examples below shows code for the RTEMS and embOS. Example code for RTEMS // rtems specific task body rtems_task oven_task(rtems_task_argument unused) { OVEN_INSTANCEDATA_T inst = OVEN_INSTANCEDATA_INIT; for (; ;) { // returns if one event was processed oven(&inst); } } // generated state machine code extern rtems_id Queue_id; uint8_t msg=NO_MSG; size_t received; rtems_status_code status; void oven(OVEN_INSTANCEDATA_T *instanceVar) { OVEN_EV_CONSUMED_FLAG_T evConsumed = 0U; /*execute entry code of default state once to init machine */ if (instanceVar->superEntry == 1U) { ovenOff(); instanceVar->superEntry = 0U; } /* action code */ /* wait for message */ status = rtems_message_queue_receive(Queue_id, (void *) &msg, &received, RTEMS_DEFAULT_OPTIONS, RTEMS_NO_TIMEOUT); if (status != RTEMS_SUCCESSFUL) error_handler(); }else{ switch (instanceVar->stateVar) { // generated state handling code … } } } Example code for embOS RTOS from Segger. // state machine instance SM_INSTANCEDATA_T instanceVar = SM_INSTANCEDATA_INIT; // Task and queue objects. static OS_STACKPTR int Stack_TASK_1[128]; /* Task stacks */ static OS_TASK TCB_TASK_1; /* Task-control-blocks */ static OS_Q MyQueue; static char MyQBuffer[100]; OS_TIMER MyTimer; char txbuf[32] ; // Routine called from the embOS RTOS to signal // a timeout. A timeout event is sent to the state // machine. Multiple timer callback functions might be created if // several timers are needed at the same time. Each one then fires an own // event. E.g. ev50ms or ev100ms static void MyTimerCallback(void) { uint8_t msg=evtTimeout; OS_Q_Put(&MyQueue, &msg, 1); } // Task blocked until a new event is present. The new event is // then sent to the state machine. static void TaskRunningStateMachine(void) { char* pData; while (1) { // waiting for new event volatile int Len = OS_Q_GetPtr(&MyQueue, (void**)&pData); volatile char msg = *pData; sm(&instanceVar, (SM_EVENT_T)msg); // call generated state machine with event OS_Q_Purge(&MyQueue); } } Sinelabore supports two basic modes of operation. Either the generated state machines react on events. Only if an event is present a transition is taken (e.g. evDoorClosed, evButtonPressed). Events are eventually send to the state machine using an event queue (see above). Alternatively transitions are triggered by boolean conditions. If a boolean condition is true a state change happens (e.g. DI0==true). The latter one is useful if binary signals should be processed like shown in these two designs (signal shaping function blocks)(PLCOpen function block). In this case the state machine runs without receiving a dedicated event. Based on the current state, conditions derived from boolean signals are used to trigger state transitions.

- cheat engine code injection
- гугл переводчик фотоперевод онлайн
- wokegepazi
- zedaha
- ecosystem matching worksheet answers