Software Development Experience

Back in college, I had many hands-on experiences through capstone projects which involved embedded programming (C/C++/LabVIEW) for platform and vehicle movement, and image recognition. In two of them, I was the team leader responsible for system architecture and task distribution, dividing software by function into small blocks for team members to implement in parallel. Meanwhile, I took charge of signal decoding, mobility, and control algorithms (>1000 lines) to manipulate motors and platform. Then, after passing the unit testing (UT), the integration testing (IT) with color recognition software developed by other team members was also successfully verified.

In graduate school, I wrote a power analysis program for the satellite in C++ (500 lines) to calculate the subsystem's power consumption and simulated the satellite's power in space. Later, I took charge of the attitude control system (ADCS) and developed the algorithm (MATLAB/Simulink, > 3000 lines) that can stabilize the satellite in four simulation hours. With the results above, I developed an embedded finite-state machine (FSM) based attitude control program in C (>2000 lines). To verify the software, I utilized the signal results from the prior software simulation and sent them to the satellite via an evaluation board (STM32) running Free-RTOS, to verify the control software's operation.

After I entered the workplace, my primary duties were to code the DVFS hardware drivers as well as the interface and control flow for higher-level users (C, > 10000 lines). I refactor code separating those less-changeable interface and control flow from drivers, and putting them in private/common folders, respectively. Under this architecture, usually only common folder required modification, and it sped up my software management cycle and testing time by a week. I implemented Hardware Verification Testing (HVT) specifically for verifying hardware functions, and feedback if any hard defects to IC designers. For software like interface and control flow, I established Unit Testing (UT) for APIs' validation and Integration Testing (IT) with upper-level SW users running under high throughput scenarios in Lab conditions. Moreover, to better maintain APIs and the naming for Users/Scenarios, Macros skills are massively used in code. Object-like and function-like Macros tend to make the code tidier, while Stringification(#) and Concatenation(##) convert arguments into string constants or merging tokens, make the naming much more flexible and adaptable.

Sheng-Hao Wu
Programming Experience

Data Structure Use Experience

Under system run-time, high-level SW users call DVFS APIs to adjust the more efficient Volt/Freq based on different scenarios. I employ data structures like queues, linked-list, and hashing in my code to record software information and hardware values for both debugging and control optimization.

Queues, follow the First-in-First-Out (FIFO) principle come in handy for software debugging with limited buffer. I implemented a 64-sized queue container (each 32bits) to record the latest 64 DVFS requests information, containing users, scenarios and time stamps, and pushing out old requests. Moreover, since DVFS hardware required settling duration and only serve one type of frequency request at a time, I designed the 3-sized queue container saving those requests that are waiting to be done, avoiding if any compulsory DVFS adjustment is missed.

Besides, to optimize DVFS control flow, I implement monitoring mechanisms that recorded all DVFS adjustments and compared them with the modem's hardware characteristics, like uplink/downlink parameters and bandwidth. Since I want to search the specific DVFS request in records as quickly as possible, I implemented a hash table to achieve the O(1) search time and chaining (linked list) to solve the hash value collision. The quicker the software search for request and compared, the faster it is to execute the optimization flow afterward and feedback to DVFS control flow and hardware.