DOI: <https://doi.org/10.15407/emodel.43.02.051>
УДК 004.451:004.354-022

П.Ю. Катін, канд. техн. наук, **О.А. Похиленко**
Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»
Україна, 03056, м. Київ, пр. Перемоги, 37,
тел. +38 067 404 93 46, e-mail: p.katin@kpi.ua;
тел. +38 063 949 83 52, e-mail: pokhilenko.alex@gmail.com

Шаблони типу Стан для створення інфраструктури системного програмного забезпечення мікроконтролерів архітектури Cortex-M у режимі реального часу для вбудованих систем

Розроблено типові програмні шаблони Стан (State pattern) в процедурному і об'єктно-орієнтованому програмуванні, які дозволяють уніфікувати вихідний код системного програмного забезпечення для мікроконтролерів архітектури Cortex-M різних виробників. Програмне забезпечення адаптовано до математичної моделі кінцевого автомата (finite-state machine (FSM)). Результати пройшли випробування на мікроконтролерах серії STM32F1xx. Застосована методика [1] дозволяє поширити отримане рішення на мікроконтролери інших виробників, що підтверджує цінність розроблених шаблонів.

К л ю ч о в і с л о в а: стандартизація і уніфікація вихідного коду, системне програмне забезпечення, програмна інфраструктура, кінцевий автомат, мікроконтролер Cortex-M, шаблон Стан, вбудована система.

Метою даного дослідження є вирішення двох завдань. Перше — створення програмної інфраструктури у вигляді типових шаблонів (patterns) для уніфікації вихідного коду мікроконтролерів архітектури Cortex-M, друге — отримання найшвидшого рішення у розроблених шаблонах для покращення швидкісних характеристик вбудованої системи (ВС). Для визначення найкращих характеристик будемо користуватися відомою методикою, описаною у роботі [1]. Для ВС, що працюють на основі операційних систем реального часу, важливо визначити потенційні обмеження за швидкістю програмної частини, щоб покращити швидкісні характеристики.

© Катін П.Ю., Похиленко О.А., 2021

Перед початком роботи потрібно визначити, що операційні системи реального часу (ОСРЧ) для мікроконтролерів (МК) можуть бути реалізовані у вигляді шаблонів на основі нескінченних циклів опитування і шаблонів програми на основі системи переривань МК.

Існують спеціалізовані ОСРЧ для МК, наприклад FreeRTOS, або повноцінні ОСРЧ, наприклад QNX, детально описані у роботі [1]. Було проведено дослідження програмних рішень ОСРЧ на основі нескінченних циклів опитування для МК на системному рівні у процедурному і об'єктно-орієнтованому програмуванні (ООП) для ВС.

Введемо обмеження. Апаратну частину ВС, що розглядаються, побудовано на основі архітектури Cortex-M великої потужності і відносно невисокої вартості. Вартість цих 32-розрядних МК архітектури Cortex-M може дорівнювати вартості деяких восьмирозрядних МК, а периферія і ядро значно швидші і досконаліші.

Програмна частина ОСРЧ для ВС, що розглядаються, побудована у вигляді нескінченного циклу опитування і може бути реалізована, по-перше, у вигляді системного програмного забезпечення через пряме програмування відповідних регістрів для управління ядром і периферією, по-друге, вона може бути реалізована з використанням універсальної бібліотеки Cortex microcontroller software Interface system (CMSIS).

У даному дослідженні варіант реалізації не має значення, тому що запропоновані шаблони типу Стан можуть бути використані для багатьох реалізацій. Це дозволяє стандартизувати (уніфікувати) результати досліджень для МК інших реалізацій архітектури ARM (Advanced RISC Machine) або використовувати ці шаблони для ВС на основі операційних систем. Відповідно забезпечується стандартизація програмних рішень ВС, побудованих на основі мов програмування C і C++. Розроблені шаблони можуть бути використані для створення бібліотек і програмної частини ВС реального часу на основі широкого спектру МК архітектури ARM і з використанням повноцінних ОСРЧ.

Аналіз літературних даних і постановка проблеми. У роботах [2—5] описано технічні рішення на основі МК Cortex-M, але не приділено уваги питанням уніфікації і стандартизації для подальшого супроводження коду і дослідженням покращення швидкості. У цих роботах не створено універсальної програмної інфраструктури, шаблонів для подальшого використання. Так, у роботі [1] описано результати дослідження універсалізації і швидкості, але при цьому не було проведено досліджень шаблону у ООП парадигмі, що суттєво обмежує програмні рішення.

В роботі [2] описано дослідження можливості збільшення швидкості програмного рішення в результаті збільшення кількості обчислювальних

ядер МК. Оскільки МК Cortex-M у більшості випадків є одноядерними, потрібно використання інших методик дослідження.

В [3] подано результати розробки контролера з використанням архітектури Cortex-M для масштабованих рішень, що потребує доопрацювання питань уніфікації вихідного коду і аналізу швидкодії. В [4] описано реалізовану систему виготовлення харчових продуктів через випаровування сировини. Результати отримано на основі МК Cortex-A, що вимагає великих витрат, а для ВС зазвичай потрібні рішення з меншими витратами, наприклад на основі Cortex-M. В [5] викладено результати рішення для управління інфраструктурою, побудовані на основі технології Інтернет, але не наведено опис типових програмних шаблонів.

Роботи, в яких досліджено програмну реалізацію шаблону Стан для МК архітектури Cortex-M, також мають недоліки щодо стандартизації і уніфікації. В роботі [6] подано реалізацію програми на основі шаблону та описано дослідження щодо збільшення швидкості обмеження, але це зроблено на основі апаратних рішень. В [7] описано результати дослідження програмної реалізації шаблону Стан в ООП для малопотужних МК в узагальненому вигляді. При цьому не вирішено задачу універсальності отриманих рішень для широкої номенклатури сучасних МК архітектури Cortex-M.

У [8] наведено результати дослідження програмної реалізації кінцевого автомата (КА) в ООП. Невирішеними залишилися питання універсальності рішень у вигляді моделі КА. В [9] запропоновано найшвидший варіант апаратної реалізації шаблону Стан. Однак при найвищій потенційній швидкості дане рішення не уніфіковано і виникають питання щодо його подальшого супроводження або пристосування для інших ВС. В [10] викладено математичну теорію опису КА, при цьому відсутні програмні приклади і зв'язок з практичними завданнями. В [11] викладено загальні описи програмних шаблонів КА, однак відсутні приклади практичних програмних рішень.

Отже, загальним недоліком робіт [6—11] є відсутність досліджень щодо потенційного обмеження швидкості типових рішень, або взагалі відсутність програмних рішень.

Мета і задачі дослідження. Розробка стандартизованого і уніфікованого програмного рішення у вигляді типового шаблону типу Стан для створення інфраструктури системного програмного забезпечення МК Cortex-M у режимі реального часу для ВС дозволить стандартизувати (уніфікувати) вихідний код для застосування у МК інших архітектур, що підтримують мови C і C++. Дослідження швидкісних характеристик дозволить їх покращити при розробці ВС.

Для вирішення поставлених завдань необхідно виконати наступне:

- розробити, протестувати і стандартизувати програми рішення типових шаблонів для МК архітектури Cortex-M без використання бібліотек CMSIS на рівні системного програмування у вигляді програмного рішення шаблону Стан;
- провести дослідження для процедурної парадигми і для ООП парадигми програмування;
- провести об'єктивну оцінку потенційної максимальної швидкості отриманих шаблонів.

Шаблон типу Стан є досить розповсюдженим шаблоном поведінки, що реалізує КА. Під час дослідження вважалося за потрібне зв'язати програмне рішення з класичною моделлю КА. Описаний у [12] КА являє собою відповідні множини і рекурентну формулу, яка найкраще підходить для опису програмного рішення:

$$\begin{aligned} z_{i+1} &= f(x_{i+1}, z_i, y_i) \\ y_{i+1} &= \varphi(x_{i+1}, z_i, y_i), \end{aligned} \quad (1)$$

де послідовність $x_0, x_1, x_2, \dots, x_n$ утворює вхідне слово (множина X); послідовність $z_0, z_1, z_2, \dots, z_k$ утворює множину станів (множина Z); послідовність $y_0, y_1, y_2, \dots, y_l$ утворює вихідне слово (множина Y). Пара (x_0, y_0) утворює початковий стан КА. Якщо (1) не залежить від Y , то перетворюється на

$$\begin{aligned} z_{i+1} &= f(x_{i+1}, z_i), \\ y_{i+1} &= \varphi(x_{i+1}, z_i), \end{aligned} \quad (2)$$

тобто у автомат Мілі. Отже (1) і (2) дають можливість подати КА у вигляді сукупності

$$(X, Y, Z, f(x_{i+1}, z_i), \varphi(x_{i+1}, z_i), Z_0), \quad (3)$$

де присутні всі програмні компоненти КА.

Розглянемо різні варіанти реалізації (3). Кінцевий автомат призначено для варіювання поведінки в залежності від внутрішнього стану програми. Якщо під час виконання програми поведінка повинна змінюватись і в коді наявні складні умовні оператори, що залежать від стану програми, то використання шаблону Стан дозволяє значно спростити такий код. При цьому стан можна трактувати як окремий об'єкт, котрий може змінюватися незалежно від інших. Використання шаблону Стан дозволяє легко додавати нові стани та переходи між ними, чого було б важко досягти використанням умовних операторів. Зазвичай використання цього шаблону є доцільним, коли стани досить часто змінюються та їх не дуже мало.

Діаграму класів реалізації шаблону Стан в ООП парадигмі наведено на рис. 1. У роботі [6], де розглянуто різні реалізації цього шаблону, показано також реалізацію у процедурному варіанті. Було вирішено відійти від типової реалізації шаблону Стан. Наприклад, для реалізації переходів між станами було використано додаткові структури, що робить модифікацію коду простішою. У розроблених програмних рішеннях контекст не створювався, а замість нього застосовано глобальну змінну з показником на поточний стан, що цілком природньо при реалізації ОСРЧ.

Рис. 1. Діаграма класів шаблону типу Стан [6]

Процедурний варіант реалізації КА без можливості додавання нових станів. Розроблений на першому етапі дослідження процедурний варіант КА без можливості додавання нових станів має простий код і являє собою рішення для ВС, що працюють за сталим алгоритмом і не потребують динамічних змін під час роботи. При розробці і тестуванні шаблонів використано типовий 32-розрядний МК STM32F103, тактова частота якого дорівнює 72 МГц, напруга живлення 3,3 В, температура в приміщенні під час тестування 27—30°C. Результати дослідження можна розповсюдити на широкий спектр МК архітектури ARM.

В даній реалізації стандартизованого шаблону Стан використано структури StateInfo і TransitionInfo. Структура StateInfo містить показник на функцію обробки станів. Формально ця структура відповідає вузлу на діаграмі станів. Створюючи нові структури StateInfo, можна збільшити кількість станів для подальшого нарощування функціоналу, але це можливо тільки до компіляції програми. Одним з прикладів покращення цієї реалізації є додавання до StateInfo функцій entry, do і exit, що дозволить повністю забезпечити формальну модель роботи технічної системи.

Структура TransitionInfo відповідає ребру орієнтованого графа станів. Вона має показники на структури попереднього і наступного станів, а також сигнал, при якому цей перехід буде виконано. У функції FSM_MakeTransition стандартизований механізм управління переходами між станами. Перший варіант управління – перехід по масиву структур, що є у вихідному коді. Другий варіант дозволяє управляти переходами при отриманні певного сигналу.

Як сигнал InputSignal можна використовувати сутність типу char або enum. В цій реалізації при отриманні некоректного сигналу у функції FSM_MakeTransition зміни станів не відбудеться, тобто непередбачені сигнали будуть проігноровані.

Рис. 2. Кінцевий автомат для тестування шаблону

Процедура додавання нових станів та переходів є дуже простою. Щоб додати новий стан, потрібно додати функцію обробки цього стану, а потім створити структуру `StateInfo` з покажчиком на цю функцію. Щоб створити новий перехід між станами, потрібно додати структуру `TransitionInfo`, вказавши сигнал, а також попередній та наступний стани, використовуючи покажчики на відповідні структури.

Для використання даної реалізації шаблону Стан потрібні дві функції. Функція `FSM_MakeTransition` призначена для переходів у інший стан при отриманні нового коректного сигналу.

Дана функція може бути розміщена у функції обробки переривань або у нескінченному циклі. Після переходу працює функція `FSM_HandleCurrent` для виконання корисної дії поточного стану. Ця функція може викликатися або один раз, наприклад перед або після функції переходу, або періодично, наприклад в нескінченному циклі.

Динамічна реалізація КА з можливістю додавання нових станів під час роботи на основі зв'язного списку. Дане рішення є складнішим за попереднє. Воно дозволяє додавати нові стани у КА під час роботи програми, що може бути корисним для систем з елементами адаптації в процесі виконання. Основна ідея полягає в тому, що у даній реалізації використовується однозв'язний список замість масиву переходів. Це дозволяє додавати нові стани і переходи під час роботи програми, використовуючи функцію `FSM_AddTransition`.

Для загальної організації роботи програми потрібен первинний покажчик на поточний стан `currentStateInfo`, масив структур переходів `transitions` і розмір цього масиву `transitionsCount`. В табл. 1 показано головний файл та файл реалізації.

На рис. 2 показано у вигляді графа достатньо простий КА для тестування шаблону. В табл. 2 наведено вихідний код з точкою входу у програму і тестом програмного рішення. Як можна побачити на рис. 2, у системі є три стани і організовано перехід між ними. Послідовність переходів відображено на рис. 3. Вона є досить простою: в нескінченному циклі виконується спочатку виклик функції корисної дії поточного стану, а потім – перехід у наступний стан. Фактично переходи між станами виконуються по колу за годинниковою стрілкою, починаючи від стану 1 (див. рис. 2). Під час тестування точки зупину ставилися усередині `while`:

Рис. 3. Діаграма послідовності зміни станів

Таблиця 1. Вміст файлів FSM.h та FSM.c

FSM.h	FSM.c
<pre> #ifndef FSM_H #define FSM_H #define InputSignal char typedef struct { void (*const handle)(void); } StateInfo; typedef struct { const StateInfo *const previous; const InputSignal input; const StateInfo *const next; } TransitionInfo; extern const StateInfo *currentStateInfo; extern const TransitionInfo transitions[]; extern const int transitionsCount; void FSM_MakeTransition(InputSignal signal); void FSM_HandleCurrent(void); #endif </pre>	<pre> #include "FSM.h" void FSM_MakeTransition(InputSignal signal) { for(int i = 0; i < transitionsCount; i++) { if(transitions[i].input == signal && transitions[i].previous == cur- rentStateInfo) { currentStateInfo = transitions[i]. next; return; } } } void FSM_HandleCurrent(void) { currentStateInfo->handle(); } </pre>

замірявся час роботи функцій FSM_HandleCurrent і FSM_MakeTransition під час покрокового виконання відповідно до розробленої методики. Код програми для тестування наведено у табл. 2.

Для створення повноцінної бібліотеки потрібно додати функцію `FSM_RemoveTransition`, щоб отримати можливість видаляти переходи (у прикладі вихідного коду її не показано). Головний файл та файл реалізації цього рішення наведено у табл. 3.

В даному рішенні є недолік у вигляді програмного циклу (`while`) у функції `FSM_MakeTransition`. Небезпека полягає в тому, що при замкненні цього списку і спробі обробити непередбачений сигнал програму буде зациклено. Для вирішення цієї проблеми пропонується додавання нових програмних елементів, які будуть перевіряти однозв'язний список на відсутність циклічності. В даному прикладі ці елементи не показано, щоб не збільшувати обсяг коду.

Таблиця 2. Вміст файлу `main.c` для першої реалізації (масив)

main.c
<pre> #include "FSM.h" void Func(void) { __nop(); } const StateInfo s1 = {Func}; // Стан 1 з функцією корисної дії Func const StateInfo s2 = {Func}; // Стан 2 з функцією корисної дії Func const StateInfo s3 = {Func}; // Стан 3 з функцією корисної дії Func const StateInfo *currentStateInfo = &s1; const TransitionInfo transitions[] = { {&s1, 0, &s2}, // Перехід зі стану 1 при отриманні сигналу 0 в стан 2 {&s2, 0, &s3}, // Перехід зі стану 2 при отриманні сигналу 0 в стан 3 {&s3, 0, &s1}, // Перехід зі стану 3 при отриманні сигналу 0 в стан 1 {&s3, 1, &s2}, // Перехід зі стану 3 при отриманні сигналу 1 в стан 2 {&s2, 1, &s1}, // Перехід зі стану 2 при отриманні сигналу 1 в стан 1 {&s1, 1, &s3} // Перехід зі стану 1 при отриманні сигналу 1 в стан 3 }; const int transitionsCount = sizeof transitions / sizeof *transitions; int main(void) { while(1) { FSM_HandleCurrent(); FSM_MakeTransition(0); } } </pre>

На відміну від попереднього рішення у даному випадку перед початком роботи (тестування) КА потрібно додати функцію ініціалізації для створення списку. Під час тестування точки зупину ставилися усередині while: замірявся час роботи функцій FSM_HandleCurrent і FSM_MakeTransition під

Таблиця 3. Вміст файлів dynamicFSM.h та dynamicFSM.c

dynamicFSM.h	dynamicFSM.c
<pre> #ifndef DYNAMICFSM_H #define DYNAMICFSM_H #define InputSignal char typedef struct { void (*const handle)(void); } StateInfo; struct TransitionInfo { const StateInfo *const previous; const InputSignal input; const StateInfo *const next; struct TransitionInfo *nextLink; }; extern const StateInfo *currentStateInfo; void FSM_AddTransition(struct TransitionInfo *transition); void FSM_MakeTransition(InputSignal signal); void FSM_HandleCurrent(void); #endif </pre>	<pre> #include "dynamicFSM.h" #define NULL ((void*)0) struct TransitionInfo *firstTransition = NULL; struct TransitionInfo *lastTransition = NULL; void FSM_AddTransition(struct TransitionInfo *transition) { if(lastTransition == NULL) { firstTransition = transition; lastTransition = transition; } else { lastTransition->nextLink = transition; lastTransition = transition; } lastTransition->nextLink = NULL; } void FSM_MakeTransition(InputSignal signal) { struct TransitionInfo *transition = firstTransition; while(transition != NULL) { if(transition->input == signal && transition->previous == currentStateInfo) { currentStateInfo = transition->next; return; } transition = transition->nextLink; } } void FSM_HandleCurrent(void) { currentStateInfo->handle(); } </pre>

час покрокового виконання. Для демонстрації роботи застосовано той самий КА, що і в попередньому рішенні (див. рис. 2).

В табл. 4 наведено точку входу в програму і тестовий код. У цьому прикладі можна побачити масив структур переходів, проте призначення його інше. Цей масив потрібен лише для того, щоб було зручно створити список у циклі, викликавши функцію `FSM_AddTransition`. Далі для роботи програми застосовується однозв'язний список. Це також потрібно для загальної стандартизації, коли треба переробити варіант, заснований на масивах, у програмне рішення, засноване на однозв'язному списку.

Реалізація КА мовою C++ з використанням окремих масивів переходів у кожному класі. Розглянемо реалізацію КА в ООП парадигмі.

Таблиця 4. Вміст файлу `main.c` для другої реалізації (список)

main.c
<pre> #include "dynamicFSM.h" void Func(void) { __nop(); } const StateInfo s1 = {Func}; // Стан 1 з функцією корисної дії Func const StateInfo s2 = {Func}; // Стан 2 з функцією корисної дії Func const StateInfo s3 = {Func}; // Стан 3 з функцією корисної дії Func const StateInfo *currentStateInfo = &s1; struct TransitionInfo transitions[] = { {&s1, 0, &s2}, // Перехід зі стану 1 при отриманні сигналу 0 в стан 2 {&s2, 0, &s3}, // Перехід зі стану 2 при отриманні сигналу 0 в стан 3 {&s3, 0, &s1}, // Перехід зі стану 3 при отриманні сигналу 0 в стан 1 {&s3, 1, &s2}, // Перехід зі стану 3 при отриманні сигналу 1 в стан 2 {&s2, 1, &s1}, // Перехід зі стану 2 при отриманні сигналу 1 в стан 1 {&s1, 1, &s3} // Перехід зі стану 1 при отриманні сигналу 1 в стан 3 }; void InitializeFSM(void) { int transitionsCount = sizeof transitions / sizeof *transitions; for(int i = 0; i < transitionsCount; i++) { FSM_AddTransition(&transitions[i]); // Додавання переходів в список } } int main(void) { InitializeFSM(); while(1) { FSM_HandleCurrent(); FSM_MakeTransition(0); } } </pre>

Для додавання нових станів і переходів в цій реалізації потрібно створювати нові нащадки абстрактного класу `AbstractState` аналогічно бібліотекам Qt. Цей абстрактний клас містить у собі методи `Handle` (еквівалент покажчика на функцію обробки `handle` у `StateInfo` у попередніх варіантах реалізацій типових шаблонів) і `MakeTransition` (еквівалент функції `FSM_MakeTransition`). Всередині цих класів містяться масиви структур переходів: кожен стан «знає» тільки свої вихідні вузли і сигнали, при яких має виконуватися перехід. У цьому рішенні потрібно створювати декілька масивів (по одному всередині кожного класу стану). Це дає можливість дотримуватися вимог інкапсуляції у парадигмі ООП.

Метод `MakeTransition` було винесено в абстрактний клас. У порівнянні з аналогічними методами попередніх рішень, цей метод є спрощеним через відсутність великої кількості перевірок. Структура `TransitionInfo` так само спростилася: вона містить лише покажчик на наступний стан і сигнал, який потрібно виконати у парадигмі ООП (табл. 5). Додавання нового стану відбувається наступним чином: створюється новий нащадок класу `AbstractState`, у нього реалізується метод `Handle` (функція об-

Таблиця 5. Вміст файлів `OOPFSM.h` та `OOPFSM.c`

OOPFSM.h	OOPFSM.c
<pre> #ifndef OOPFSM_H #define OOPFSM_H #define InputSignal char class AbstractState; typedef struct { const InputSignal input; const AbstractState *const next; } TransitionInfo; class AbstractState { private: const TransitionInfo *const tran- sitions; const int transitionsCount; public: virtual void Handle(void) const = 0; virtual void MakeTransi- tion(InputSignal signal) const; explicit AbstractState(const TransitionInfo transitions[], const int count); ~AbstractState(); }; extern const AbstractState *currentState; #endif </pre>	<pre> #include "OOPFSM.h" AbstractState::AbstractState(const TransitionIn- fo trans[], const int count) : transitions(trans), transitionsCount(count) {} void AbstractState::MakeTransition(InputSignal signal) const { for(int i = 0; i < transitionsCount; i++) { if(transitions[i].input == signal) { currentState = transitions[i].next; return; } } } AbstractState::~~AbstractState() {delete[] transitions;} </pre>

робки або корисної дії цього стану), створюється екземпляр цього нового класу і йому через конструктор передається масив переходів, які можна виконати з цього вузла (стану). Для переходів між станами і виконання корисної дії можна або безпосередньо працювати з `currentState` — покажчиком на поточний клас стану, або створити «обгортку», де будуть відокремлені функції `FSM_MakeTransition` і `FSM_HandleCurrent`.

В табл. 6 подано процес тестування і точка входу. У цьому прикладі виклик методів відбувався через `currentState` безпосередньо, без викорис-

Таблиця 6. Вміст файлу `main.cpp` для третьої реалізації (ООП)

main.cpp
<pre> #include "OOPFSM.h" extern const TransitionInfo s1Transitions[2]; // Масив переходів зі стану 1 extern const TransitionInfo s2Transitions[2]; // Масив переходів зі стану 2 extern const TransitionInfo s3Transitions[2]; // Масив переходів зі стану 3 class State1 : public AbstractState { public: void Handle(void) const { __nop(); }; explicit State1(const TransitionInfo trans[], const int count): AbstractState(trans, count) {}; } s1(s1Transitions, 2); // Стан 1 class State2 : public AbstractState { public: void Handle(void) const { __nop(); }; explicit State2(const TransitionInfo trans[], const int count): AbstractState(trans, count) {}; } s2(s2Transitions, 2); // Стан 2 class State3 : public AbstractState { public: void Handle(void) const { __nop(); }; explicit State3(const TransitionInfo trans[], const int count): AbstractState(trans, count) {}; } s3(s3Transitions, 2); // Стан 3 const AbstractState *currentState = &s1; // Заповнення масивів переходів для станів 1, 2 і 3 відповідно: const TransitionInfo s1Transitions[] = {{0,&s2}, {1,&s3}}; //при сигналі 0 в стан 2, при 1 – в 3 const TransitionInfo s2Transitions[] = {{0,&s3}, {1,&s1}}; //при сигналі 0 в стан 3, при 1 – в 1 const TransitionInfo s3Transitions[] = {{0,&s1}, {1,&s2}}; //при сигналі 0 в стан 1, при 1 – в 2 int main(void) { while(1) { currentState->Handle(); currentState->MakeTransition(0); } } </pre>

тання додаткової обгортки. Заміри виконання методів було зроблено так само за допомогою покрокового виконання і обчислення різниці часу між початком і завершенням виконання методу.

Результати оцінки швидкості розроблених шаблонів. Для визначення часу на переключення (перехід) між станами застосовано відому методику [1] оцінки швидкості на основі підрахунку тактів у режимі покрокового налагодження МК в середовищі розробки Keil. Особливістю методики є одиниці оцінки часу роботи програмного рішення. Оцінка часу роботи програмного рішення проводилася з використанням можливостей середовища розробки Keil. При цьому вимірювався не час роботи між кроками програми, а здійснювався підрахунок кількості тактів, записаних в регістрі Cycle Count (DWT_CYCCNT) register. Далі будемо називати це значення кількістю тактів. Його можна визначити в ході покрокового налагодження в середовищі розробки Keil.

Такий підхід дає можливість узагальнити результати оцінки швидкості програмного рішення на МК Cortex-M інших виробників. Оскільки МК Cortex-M можуть мати різну тактову частоту, оцінка часу, отримана для одного МК, не може бути узагальнена на МК різних виробників. А оцінка кількості тактів дозволяє порівняти швидкість для різних програмних рішень і узагальнити її для МК Cortex-M різних виробників, що підтримують бібліотеки CMSIS в середовищі Keil.

Статистична оцінка кількості тактів не проводилася, тому що в багаторазових експериментах значення States для однакових точок зупину не змінювалося. Результати дослідження подано у табл. 7. Було зроблено заміри для функції додавання FSM_AddTransition при реалізації на основі однозв'язного списку. Виявилось, що додавання займає 55 тактів для першого елемента та 53 для наступних.

Таблиця 7. Результати порівняння швидкості переходів між станами

Реалізація	Перехід (MakeTransition)	Виконання (HandleCurrent)	Пояснення
Масив	$106 + i * 62$	41	i – індекс переходу в масиві (від 0 до $n-1$), тобто перехід $s_3 - 0- > s_1$ буде займати $106 + 2 * 62 = 230$, тому що в масиві він має індекс 2
Список	$58 + i * 30$	41	i – індекс в однозв'язному списку (від 0 до $n-1$)
ООП	$79 + i_j * 25$	33	i_j – індекс переходу в масиві j (масив переходів зі стану j), тобто перехід $s_1 - 1- > s_3$ буде займати $79 + 1 * 25 = 104$, тому що цей перехід має індекс 1 в масиві переходів зі стану 1

Розроблені програмні шаблони Стан стандартизовано і уніфіковано для програмних рішень ВС у трьох варіантах:

1) процедурний варіант реалізації КА ВС без можливості додавання нових станів, як найпростіше рішення;

2) процедурний варіант і динамічна реалізація КА ВС з можливістю додавання нових станів під час роботи на основі зв'язного списку;

3) реалізація КА у ООП парадигмі програмування ВС мовою C++. На відміну від відомих класичних рішень шаблону Стан в даному рішенні використовуються окремі масиви переходів у кожному класі і воно може бути розроблено на основі попередніх розглянутих рішень, написаних мовою С.

Результат оцінки швидкості переходів між станами суперечить висновкам роботи [1], у якій стверджується, що швидкість переходу між станами у процедурному варіанті більша, ніж реалізація у парадигмі ООП на основі якісного аналізу. Проте результати вимірювання, отримані у даному дослідженні, засвідчили, що реалізація ООП (див. табл. 7) є найшвидшою за виконанням та потребує найменше часу, необхідного на перехід. Це обґрунтовано тим, що у реалізації в парадигмі ООП масив станів має декомпозицію на кожний стан. Вдосконалюючи реалізацію у парадигмі ООП, можна замінити масиви на однозв'язні списки, що дозволить динамічно змінювати переходи між станами. Також потрібна розробка реальної програми з використанням отриманих стандартизованих рішень і проведення тестування.

Висновки

Отримані програмні рішення у вигляді типового шаблону Стан дозволяють стандартизувати програмну частину ВС на системному рівні і є основою створення інфраструктури системного програмного забезпечення ВС на основі МК Cortex-M у режимі реального часу.

Результати оцінки швидкості роботи розроблених трьох програмних рішень для ВС засвідчили, що реалізація у ООП парадигмі програмування є найшвидшою за виконанням та потребує найменше часу, необхідного на перехід.

У дослідженні використано комплект налагодження для 32-розрядного МК архітектури ARM (Cortex-M) типу STM32F103 без підключення зовнішніх пристроїв. У ході експериментів периферійні пристрої МК не використовувалися для організації аналогічних досліджень МК архітектури ARM (Cortex-M) інших виробників. Тому структурну схему не наведено. Отже, такі дослідження можна провести з іншими типами МК для узагальнення результатів. Отримані стандартизовані шаблони доцільно використовувати у випадках обмеженості вбудованої пам'яті МК

Random Access Memory, для збільшення швидкості ВС, у тому числі з використанням повноцінних ОСРЧ, і для виконавчих систем з адаптацією і штучним інтелектом.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Chmelov V.O., Katin P.Y. and Shemaev V.M. Development of typical «State» software patterns for Cortex-M microcontrollers in real time// *Eastern-European Journal of Enterprise Technologies*, 2020, 3(9 (105)), pp. 29—38.
2. Saini P., Bansal A. and Sharma A. Time critical multitasking for multicore microcontroller using XMOS® kit// *International Journal of Embedded systems and Applications (IJESA)*, 2015, 5(1), pp. 1-18.
3. Chen Z., Chen J. and Zhou S. Embedded electronic scale measuring system based on STM32 single chip microcomputer// *Proceedings, 2019 Chinese Automation Congress*, Hangzhou, China, November 22-24, 2019, pp. 3062—3065.
4. Катін П.Ю., Бессмертний П.С. Використання високопродуктивних мікроконтролерів для підвищення економічної ефективності виробництва джему // *Стандартизація, сертифікація, якість*, 2019, 3(115), с. 69—77.
5. Zhu W., Wang Z. and Zhang Z. Renovation of Automation System Based on Industrial Internet of Things: A Case Study of a Sewage Treatment Plant// *Sensors*, 2020, 20(8), 2175.
6. Gamma E., Helm R., Johnson R. and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software* // Addison-Wesley, 1994.
7. Katin P.Y. Development of variant of software architecture implementation for low-power general purpose microcontrollers by finite state machines // *EUREKA: Physics and Engineering*, 2017, 3, pp. 49—55.
8. Solodovnikov A. Developing method for assessing functional complexity of software information system// *Ibid*, 2016, 5, pp. 35—44.
9. Dietrich C., Hoffmann M. and Lohmann D. Back to the Roots: Implementing the RTOS as a Specialized State Machine. *OSPERT 2015*, Lund, Sweden, July 7, 2015, pp. 7—12.
10. Beynon W. On the structure of free finite state machines// *Theoretical Computer Science*, 1980, 11, pp. 167—180.
11. Adamczyk P. The Anthology of the Finite State Machine Design Patterns, 2013. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.838&rep=rep1&type=pdf>.
12. Блох А.И. Граф-схемы и их применение. Минск: Высшая школа, 1975, 294 с.

Отримано 28.12.20;
після доопрацювання 25.02.21

REFERENCES

1. Chmelov, V.O., Katin, P.Y. and Shemaev, V.M. (2020), "Development of typical "State" software patterns for Cortex-M microcontrollers in real time", *Eastern-European Journal of Enterprise Technologies*, Vol. 3, no. 9 (105), pp. 29-38.
2. Saini, P., Bansal, A. and Sharma, A. (2015), "Time critical multitasking for multicore microcontroller using XMOS® kit", *International Journal of Embedded systems and Applications (IJESA)*, Vol. 5, no. 1, pp. 1-18.
3. Chen, Z., Chen, J. and Zhou, S. (2019), "Embedded electronic scale measuring system based on STM32 single chip microcomputer", *CAC 2019, Proceedings – 2019 Chinese Automation Congress*, Hangzhou, China, November 22-24, 2019, pp. 3062-3065.

4. Katin, P.Y. and Bessmertnyy, R.S. (2019), "Use of high-performance microcontroller for improving economic efficiency of jam production", *Standartyzatsiya, sertyfikatsiya, yakist*, Vol. 3, no. 115, pp. 69-77.
5. Zhu, W., Wang, Z. and Zhang, Z. (2020), "Renovation of Automation System Based on Industrial Internet of Things: A Case Study of a Sewage Treatment Plant", *Sensors*, Vol. 20, no. 8, 2175.
6. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, Massachusetts.
7. Katin, P.Y. (2017), "Development of variant of software architecture implementation for low-power general purpose microcontrollers by finite state machines", *EUREKA: Physics and Engineering*, Vol. 3, pp. 49-55.
8. Solodovnikov, A. (2016), "Developing method for assessing functional complexity of software information system", *Ibid*, Vol. 5, pp. 35-44.
9. Dietrich, C., Hoffmann, M. and Lohmann, D. (2015), "Back to the Roots: Implementing the RTOS as a Specialized State Machine", *Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Proceedings of OSPERT 2015*, Lund, Sweden, July 7, 2015, pp. 7-12.
10. Beynon, W. (1980), "On the structure of free finite state machines", *Theoretical Computer Science*, Vol. 11, pp. 167-180.
11. Adamczyk, P. (2003), "The Anthology of the Finite State Machine Design Patterns", available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.838&rep=rep1&type=pdf> (accessed February 20, 2021).
12. Blokh, A.Sh. (1975), *Graf-skhemy i ikh primeneniye*. [Graph schemes and their application], Vysheyschaya shkola, Minsk, Belarus.

Received 28.12.20;
After revision 25.02.21

P.Y. Katin, O.A. Pokhylenko

TYPICAL "STATE" SOFTWARE PATTERNS FOR CREATING CORTEX-M MICROCONTROLLER SYSTEM SOFTWARE INFRASTRUCTURE IN REAL TIME EMBEDDED SYSTEMS

To standardize and unify the source code and mathematical modeling of system software, it is advisable to use standard templates (patterns). They create a system software infrastructure for later use for program development. This is especially important for real-time operating systems, where you can pre-determine possible speed limits for embedded programs. For this purpose, for embedded systems based on microcontrollers of the Cortex-M architecture, standard software patterns State have been developed in the procedural and OOP paradigm. The developed patterns allow to unify the source code of the system software for microcontrollers of the Cortex-M architecture of different manufacturers. Research has been conducted to determine the speed limit of real-time operations. The software is adapted to the mathematical model of a finite state machine. The results were tested on microcontrollers of the STM32F1xx series. This allows to extend the solution to the MC of other manufacturers, which confirms the practical value of the developed patterns.

Keywords: standardization and unification of source code, system software, software infrastructure, finite state machine, Cortex-M microcontroller, State pattern, embedded system.

КАТІН Павло Юрійович, канд. техн. наук, доцент кафедри автоматики і управління в технічних системах Національного технічного університету України «Київський політехнічний інститут ім. Ігоря Сікорського». У 1994 р. закінчив Київський військовий інститут управління і зв'язку, у 2015 р. — Національний технічний університет України «Київський політехнічний інститут ім. Ігоря Сікорського». Область наукових досліджень — математична формалізація і розробка програмного забезпечення для вбудованих систем, комп'ютерних систем, інтегрованих інформаційних систем.

ПОХИЛЕНКО Олександр Андрійович, студент кафедри автоматики і управління в технічних системах Національного технічного університету України «Київський політехнічний інститут ім. Ігоря Сікорського». Область наукових досліджень — математична формалізація і розробка програмного забезпечення для вбудованих систем, комп'ютерних систем, інтегрованих інформаційних систем.