<https://barenakedembedded.com/how-to-use-cpp-with-stm32-hal/>

Как использовать C ++ с STM32 HAL

STM32 HAL (уровень аппаратной абстракции) ускоряет циклы разработки, упрощает переключение между процессорами STM32 и располагает огромным сообществом поддержки. HAL является частью набора инструментов STM32CubeIDE, который позволяет пользователям автоматически генерировать код для различных периферийных устройств STM32. HAL можно рассматривать как высокоабстрагированную библиотеку, которая практически универсальна для процессоров STM32. STM32 HAL может быть очень полезным инструментом, но есть и недостатки. Использование HAL может привести к раздуванию кода, создать ложное представление об аппаратном обеспечении и может **генерировать только встроенный код C**.

В этом руководстве на высоком уровне показано, как управлять графическими процессорами HAL и таймерами с C++. Для этого создается “цикл событий C++”, который будет действовать как точка входа в C++, вызываемая из main.c. Затем создается класс-оболочка C++ GPIO, чтобы управлять выводами GPIO через HAL из цикла событий C++. После этого я дам краткое описание обработки прерываний для модулей таймера и UARTS / SPI с использованием HAL в C++. Весь код, используемый в этом руководстве, можно найти в следующем репозитории GitHub, [stm32-hal-with-cpp](https://github.com/BradenSunwold/stm32-hal-with-cpp).

Цикл событий C ++

“Цикл событий” будет действовать как наша точка входа в C++ при смешивании C++ с С в STM32. Поскольку HAL генерирует файл main.c на языке С, нам нужно вызвать функцию C++, чтобы использовать возможности C ++.

Создайте свои файлы проекта на С++, например, “EventLoop.hpp” и “EventLoop.cpp”. В EventLoop.hpp создайте две версии функции EventLoop() как на C, так и на C ++, как показано ниже.

EventLoop.hpp:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | #ifndef EVENTLOOP\_HPP\_  #define EVENTLOOP\_HPP\_    #include <stdint.h>  #include <stdio.h>    **void** EventLoopCpp();   // Cpp function to call into main event loop    #ifdef \_\_cplusplus  **extern** "C"  {  #endif  **void** EventLoopC();  // C function to call into Cpp event loop from main  #ifdef \_\_cplusplus  }  #endif    #endif /\* EVENTLOOP\_HPP\_ \*/ |

Extern “C” позволяет создавать код C, встроенный в файл C ++. Main.c вызовет версию функции на языке C, которая размещена в файле на С++, а уже версия C вызовет версию функции на C ++. Фишка в том, чтобы разместить код на С внутри файла с кодом на С++. Компилятор, обрабатывая код на С++, корректно обработает и код на С, соответственно, вызовы функций и обращения к переменным также будут корректно обработаны. Обратите внимание на ключевое слово «\_\_cplusplus» - в нём два(!) нижних подчёркивания.

EventLoop.cpp:

|  |  |
| --- | --- |
| 2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #include "EventLoop.hpp"    // Main Cpp event loop to run application  **void** EventLoopCpp()  {    }    // Define all C function calls from main.c below  **extern** "C"  {  **void** EventLoopC()      {          EventLoopCpp();      }  } |

Этот обходной способ вызова функции C++ EventLoop() из кода на языке С необходим, поскольку main.c не может вызывать функцию C++ напрямую.

Определение классов в цикле событий C++

Теперь, когда у нас есть точка входа C++ из main.c, нам нужно проверить, действительно ли здесь будет выполняться специфичный для C++ код. Для этого создайте пустой класс для тестирования. Мы создадим тестовый класс с фиктивным конструктором, в котором, в конечном итоге, будет реализована наша функциональность blink. Создайте “CppLedBlink.hpp” и “CppLedBlink.cpp”.

CppLedBlink.hpp:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #ifndef CPPLEDBLINK\_HPP\_  #define CPPLEDBLINK\_HPP\_    **class** CppLedBlink  {  **public**:      CppLedBlink();  };    #endif /\* CPPLEDBLINK\_HPP\_ \*/ |

CppLedBlink.cpp:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #include "CppLedBlink.hpp"    // Dummy constructor for testing C++  CppLedBlink::CppLedBlink()  {  **int** test = 1;  } |

Теперь следующий пункт, возможно, самое важное, что следует иметь в виду при смешивании C++ с STM32 HAL. От того, как вы включите (include) этот класс в файлы EventLoop, зависит, будет ли ваша программа работать или нет. Если вы попытаетесь включить (include) CppLedBlink.hpp, в котором определён класс CppLedBlink, обычным способом через заголовочный файл EventLoop.hpp, вы получите сообщение об ошибке “неизвестное имя типа ‘class'” (“unknown type name ‘class'”). Это связано с тем, что main.c включает (include) EventLoop.hpp, поэтому он будет скомпилирован с помощью компилятора C вместо компилятора C++. Так как CppLedBlink.hpp включен в EventLoop.hpp, то и CppLedBlink.hpp так же будет обработан компилятором С. А вот EventLoop.cpp, в свою очередь, будет скомпилирован с помощью компилятора C++, поэтому любой специфичный для C++ код должен быть расположен внутри EventLoop.cpp.

Ошибка в CppLedBlink.hpp:

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

Хитрость в том, что вы должны включить (include) CppLedBlink.hpp в файл EventLoop.cpp вместо EventLoop.hpp. Теперь внутри EventLoop.cpp отредактируйте функцию EventLoopCpp() для создания экземпляра объекта CppLedBlink. Ошибок быть не должно.

EventLoop.cpp:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #include "EventLoop.hpp"    // Main Cpp event loop to run application  **void** EventLoopCpp()  {      CppLedBlink testing;  }    // Define all C function calls from main.c below  **extern** "C"  {  **void** EventLoopC()      {          EventLoopCpp();      }  } |

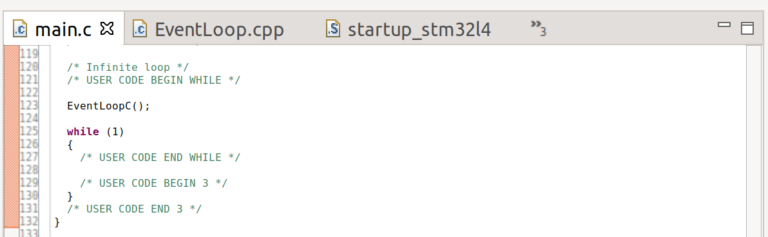
Наконец, убедитесь, что EventLoop.hpp включен в main.c в пределах /\* USER CODE BEGIN Includes \*/ и /\* USER CODE END Includes \*/. Затем вызовите функцию EventLoopC() над основным циклом while(1), как показано ниже.

Include EventLoop.hpp in main.c:

Изображение выглядит как текст, снимок экрана, дисплей, число

Автоматически созданное описание

Вызов EventLoopC() из main.c



Перенос HAL GPIO

Теперь, когда мы запустили C ++ с STM32, мы хотим использовать HAL. Для этого мы создадим два класса C ++ для переноса функций GPIO HAL. Первый класс определит вывод GPIO и будет иметь функции-члены для переключения, чтения, очистки и установки вывода GPIO. Эти функции-члены используют GPIO HAL для выполнения каждой задачи. Второй класс определяет распиновку для конкретного проекта и будет содержать элементы для всех необходимых выводов GPIO.

Определения выводов HAL GPIO состоят из “pin”, который просто определяется как uint16\_t, и “port”, который определяется как структура “GPIO\_TypeDef”. Следовательно, нашему классу вывода GPIO потребуется uint16\_t и переданный в него указатель на GPIO\_TypeDef. GPIO\_TypeDef и связанные с ним функции GPIO HAL включены в файл “stm32l4xx\_hal.h”. Этот файл будет назван в соответствии с тем, какой процессор или комплект разработки используется.

GpioPin.hpp:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | #ifndef GPIOPIN\_HPP\_  #define GPIOPIN\_HPP\_    #include "stm32l4xx\_hal.h"    **class** GpioPin  {  **public**:      GpioPin(**uint16\_t** pin, GPIO\_TypeDef\* port);   // Constructor    **bool** Read();  **void** Set();  **void** Clear();  **void** Toggle();    **private**:  **uint16\_t** mGpioPin;      GPIO\_TypeDef\* mGpioPort;  };    #endif /\* GPIOPIN\_HPP\_ \*/ |

GpioPin.cpp:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | #include "GpioPin.hpp"    GpioPin::GpioPin(**uint16\_t** pin, GPIO\_TypeDef\* port)      :      mGpioPin(pin),      mGpioPort(port)  {  }    **bool** GpioPin::Read()  {  **return** HAL\_GPIO\_ReadPin(mGpioPort, mGpioPin);  }    **void** GpioPin::Set()  {      HAL\_GPIO\_WritePin(mGpioPort, mGpioPin, GPIO\_PIN\_SET);  }    **void** GpioPin::Clear()  {      HAL\_GPIO\_WritePin(mGpioPort, mGpioPin, GPIO\_PIN\_CLEAR);  }    **void** GpioPin::Toggle()  {      HAL\_GPIO\_TogglePin(mGpioPort, mGpioPin);  } |

Выводы и порты HAL GPIO для автоматически сгенерированного проекта расположены в main.h. Таким образом, после настройки вывода GPIO по мере необходимости в представлении .ioc, мы можем перенести его из main.h в наш проект C ++ через класс распиновки (Pinout). В классе Pinout мы определяем любые контакты GPIO, которые нам нужны, как объекты GpioPin (см. Выше). Это будут общедоступные элементы, чтобы мы могли получить доступ к каждому элементу GPIO из-за пределов области видимости класса. Затем мы передаем любые GPIO, которые мы хотим, в наш список инициализаторов конструктора из main.h.

CppBlinkPinout.hpp

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #ifndef CPPBLINKPINOUT\_HPP\_  #define CPPBLINKPINOUT\_HPP\_    #include "GpioPin.hpp"  #include "main.h"    **class** CppBlinkPinout  {  **public**:      CppBlinkPinout();        // Define all wanted GPIO pins for specific project below      GpioPin mTestLed;  };    #endif /\* CPPBLINKPINOUT\_HPP\_ \*/ |

CppBlinkPinout.cpp

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | #include "CppBlinkPinout.hpp"    CppBlinkPinout::CppBlinkPinout()      :      // Wrap all GPIO pins / ports into project from main.h      mTestLed(LED2\_Pin, LED2\_GPIO\_Port)  {  } |

Собрать все это воедино

Последний шаг - собрать все воедино, чтобы показать фактическое мигание светодиода в нашем проекте C ++ с STM32. Вернитесь к классу CppLedBlink и добавьте частный элемент CppBlinkPinout. Это будет использоваться для мигания светодиода в классе CppLedBlink. На данный момент для переключения светодиода используется простая задержка цикла for, но также можно использовать функцию HAL\_Delay(). Мы также кратко рассмотрим, как интегрировать модули таймера в проект C ++, которые также могут быть использованы для создания задержки.

CppLedBlink.hpp:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #ifndef CPPLEDBLINK\_HPP\_  #define CPPLEDBLINK\_HPP\_    #include "CppBlinkPinout.hpp"    **class** CppLedBlink  {  **public**:      CppLedBlink();    **private**:      CppBlinkPinout mPinout;  };    #endif /\* CPPLEDBLINK\_HPP\_ \*/ |

CppLedBlink.cpp:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #include "CppLedBlink.hpp"    // Dummy constructor for testing C++  CppLedBlink::CppLedBlink()  {  **while**(1)      {  **for**(**int** i = 0; i < 1000000; i++);          mPinout.mTestLed.Toggle();      }    } |

Идем дальше

При смешивании C ++ с STM32 может понадобиться гораздо больше периферийных устройств, помимо только GPIO. В этом разделе будет дан обзор того, как интегрировать модули таймера HAL и UART / SPI с C ++.

Использование этих периферийных устройств с помощью HAL и C ++ требует реализации функции инициализации интерфейса, которая вызывается из main.c и связывает все компоненты HAL со стороны C на сторону C ++.

# Таймеры

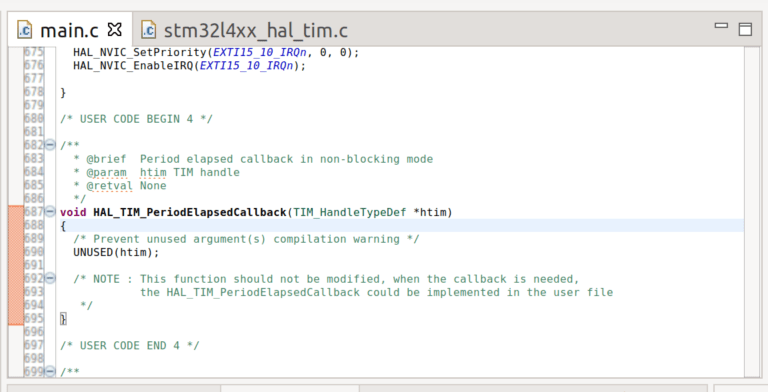
Аппаратные таймеры постоянно выполняются и настроены на прерывание при переполнении или событиях “периода”. Эти прерывания будут обрабатываться через HAL / C-сторону приложения. При прерывании переполнения переменная переполнения таймера будет увеличена, чтобы отслеживать, сколько раз наш таймер переполнялся. Мы создадим функции взаимодействия в main.c, которые позволят считывать аппаратный таймер и переменные переполнения. Эти функции взаимодействия будут переданы стороне C ++ в виде указателей на функции через функцию “инициализации”, вызываемую из main.c.

Используйте представление конфигурации .ioc для инициализации аппаратного таймера. В этом примере будет использоваться таймер 2, установленный как 32-разрядный таймер, работающий на частоте 1 МГц. Поскольку внутренние часы для STM32L475 работают на частоте 80 МГц, предварительное масштабирование будет установлено на 80, чтобы создать тактовую частоту 1 МГц. Фактическое значение будет установлено в (значение предварительного масштабирования минус 1) или 79. Режим подсчета будет оставлен в режиме “Вверх”, а период будет оставлен на максимальном значении.

Прерывания STM32 HAL используют предопределенные функции обратного вызова, которые могут быть переопределены в main.c для достижения некоторой желаемой функциональности при прерывании. Эти функции определены как “слабые”, поскольку они предназначены для переопределения пользователем. Функции обратного вызова прерывания по таймеру расположены по адресу Drivers/STM32L4xx\_HAL\_Drivers/Src/stm32l4xx\_hal\_tim.c.

Сейчас просто скопируйте функцию HAL\_TIM\_PeriodElapsedCallback() из этого файла в main.c и удалите тег \_\_weak, как показано ниже.

Period Elapsed Callback in main.c User Code Section 4:

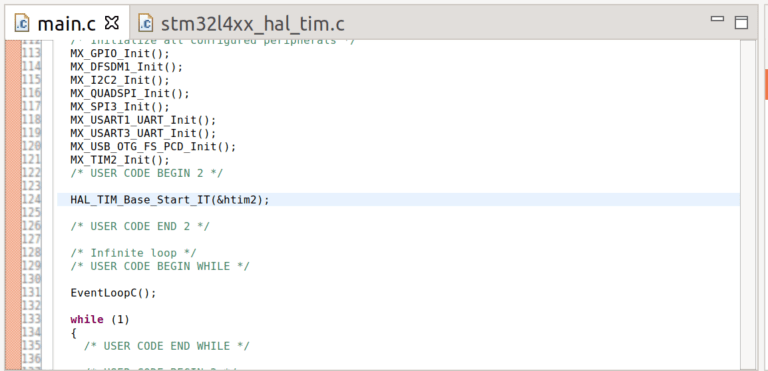


В main.c создайте новую глобальную переменную timerOverflow и увеличьте ее в обратном вызове таймера.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | // Above main  **volatile** **uint32\_t** timerOverflow = 0;    // Callback re-defined within main.c  **void** HAL\_TIM\_PeriodElapsedCallback(TIM\_HandleTypeDef \*htim)  {      /\* Prevent unused argument(s) compilation warning \*/      UNUSED(him);        // Ensure that it was timer 2 that interrupted  **if**(htim == htim2)      {          timerOverflow++;      }  } |

После настройки прерывания по таймеру и обратного вызова нам нужно запустить таймер, а затем создать интерфейс на стороне C ++.

Запуск таймеров в main.c:



Создайте функцию в main.c для считывания значения аппаратного таймера и переменной переполнения таймера.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **uint32\_t** ReadHardwareTimer()  {  **uint32\_t** count = htim2.Instance->CNT;  **return** count;  }    **uint32\_t** ReadTimerOverflow()  {  **return** timerOverflow;  } |

Затем, вернувшись в C ++ EventLoop, создайте функцию инициализации, в которой мы сможем передавать функции из main.c на сторону C ++ через указатели на функции.

EventLoop.hpp:

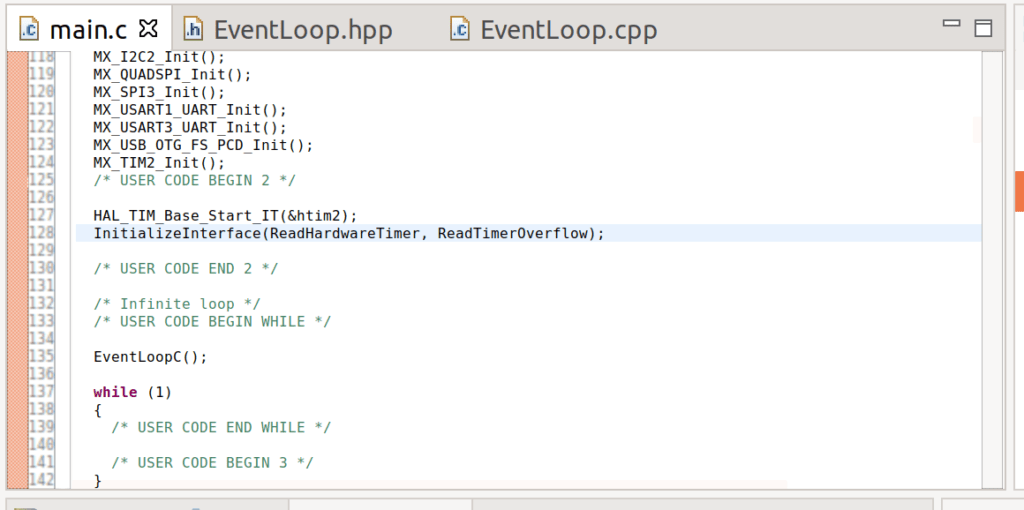
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | #ifndef EVENTLOOP\_HPP\_  #define EVENTLOOP\_HPP\_    #include <stdint.h>  #include <stdio.h>    // Define func pointers for reading hw timers and timer overflows  **typedef** uin32\_t (\*ReadHwTimer)();  **typedef** **uint32\_t** (\*ReadOverflow)();    **void** EventLoopCpp();   // Cpp function to call into main event loop    #ifdef \_\_cplusplus  **extern** "C"  {  #endif  **void** EventLoopC();   // C function to call into Cpp event loop from main.  **void** InitializeInterface(ReadHwTimer readHwTimer, ReadOverflow readTimerOverFlow);  #ifdef \_\_cplusplus  }  #endif    #endif /\* EVENTLOOP\_HPP\_ \*/ |

EventLoop.cpp:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | #include "EventLoop.hpp"    // Main Cpp event loop to run application  **void** EventLoopCpp()  {      CppLedBlink testing;  }    // Define all C function calls from main.c below  **extern** "C"  {  **void** EventLoopC()      {          EventLoopCpp();      }    **void** InitializeInterface(ReadHwTimer readHwTimer, ReadOverflow readTimerOverFlow)      {          /\*          \* Can now link these function pointers to any C++ class for use.          \* For example, Could pass them into a static global system time          \* class where they would be used to calculate the ongoing system          \* time from system startup.          \*/      }  } |

Наконец, вызовите функцию инициализации в main.c перед вызовом цикла событий C ++ и передайте функции ReadHardwareTimer() и ReadTimerOverflow(). Теперь эти указатели на функции можно передавать любым глобальным объектам C ++ в файле EventLoop.cpp . Например, они могут быть переданы в класс системного времени, который может считывать их для вычисления общего времени, прошедшего с момента запуска программы.

Вызов интерфейса инициализации в main.c:



# UART/ SPI

Интерфейсы UART и SPI HAL используют структуру UART\_HandleTypeDef или SPI\_HandleTypeDef, которая определяет такие вещи, как размеры буфера, коды ошибок и указатели функций на конкретные процедуры обслуживания прерываний (ISR). Когда прерывание запускается для UART или SPI, соответствующий ISR вызывается через этот указатель функции в коде HAL более низкого уровня. Благодаря этому можно написать пользовательский ISR и связать UART\_HandleTypeDef / SPI\_HandleTypeDef с этим новым пользовательским ISR.

Хорошим примером этого является ISR UART Tx. Пользовательский Tx UART ISR может быть реализован во внешнем блоке C, чтобы любые байты, содержащиеся в определенном буфере, отправлялись по линии UART Tx перед отключением прерывания Tx. Тогда аналогично реализации указателя функции timer ReadHardwareTimer() или ReadTimerOverflow(), описанной выше, указатель функции может использоваться для включения прерывания UART Tx в любое время, когда этот конкретный буфер записывается на стороне C ++. Создать пользовательский ISR для UARTS или SPI не слишком сложно, поскольку встроенные функции HAL можно использовать в качестве ссылок. Посмотрите на HAL\_UART\_Transmit\_IT() и HAL\_UART\_Recieve\_IT() в качестве примеров, чтобы начать.