# <https://support.touchgfx.com/docs/development/ui-development/touchgfx-engine-features/backend-communication>

# Внутренняя коммуникация

В большинстве приложений пользовательский интерфейс должен быть каким-то образом подключен к остальной части вашей системы, а также отправлять и получать данные. Это может быть взаимодействие с аппаратной периферией (данные датчиков, A / D преобразования, последовательная связь, ...) или взаимодействие с другими программными модулями.

В этой статье описываются рекомендуемые решения для реализации этого подключения.

Первый метод - это ["быстрый и грязный"](https://support.touchgfx.com/docs/development/ui-development/touchgfx-engine-features/backend-communication#sampling-from-gui-task) подход, в первую очередь предназначенный для прототипирования, тогда как [второй метод](https://support.touchgfx.com/docs/development/ui-development/touchgfx-engine-features/backend-communication#sampling-from-secondary-task) - это архитектурно обоснованный способ правильного соединения пользовательского интерфейса с остальными компонентами реального приложения.

В конце этой статьи мы даем ссылки на примеры использования обоих методов.

Класс Model

Все приложения TouchGFX имеют класс Model, который помимо хранения информации о состоянии пользовательского интерфейса также предназначен для функционирования в качестве интерфейса к вашей окружающей системе. Под этим мы подразумеваем как аппаратную периферию, так и взаимодействие с другими задачами операционной системы в вашей системе. Обычно не рекомендуется использовать доступ к другим программным модулям или оборудованию в отдельных классах представления.

##### ДАЛЬНЕЙШЕЕ ЧТЕНИЕ

Чтобы узнать больше о модели: [шаблон MVP](https://support.touchgfx.com/docs/development/ui-development/software-architecture/model-view-presenter-design-pattern)

Класс Model хорошо подходит для размещения любого такого кода интерфейса, потому что:

1. Класс Model имеет tick() функцию, которая автоматически вызывается в каждом кадре и может быть реализована для поиска событий из других подмодулей и реагирования на них.
2. Класс Model содержит указатель на ваш текущий активный презентатор, чтобы иметь возможность уведомлять пользовательский интерфейс о входящих событиях. Этот указатель содержится в ModelListener.

# Взаимодействие с системой

Существует два способа взаимодействия с окружающей системой: либо путем выборки непосредственно из задачи GUI, либо путем выборки из вторичной задачи.

## Выборка из задачи GUI

Наилучший способ взаимодействия с окружающей системой зависит от того, как часто вам нужно выполнять сэмплирование, насколько это трудоемко и насколько критично по времени.

Если ваши требования в этом отношении снисходительны, самый простой подход - просто попробовать окружающую систему непосредственно в Model::tick функции.

Если выборка происходит реже, чем ваша частота кадров (обычно около 60 Гц), вы можете просто добавить счетчик и выполнять выборку только через каждый N-й тик. При выполнении этого способа ваша операция выборки должна быть несколько быстрой (обычно 1 мс или меньше), в противном случае ваша частота кадров начнет снижаться, поскольку выборка выполняется в контексте задачи GUI и приведет к задержке рисования кадра.

## Выборка из вторичной задачи

В качестве альтернативы, если нежелательно помещать взаимодействие с окружающей системой непосредственно в контекст задачи (task) GUI, вы можете создать новую задачу операционной системы, ответственную за выполнение выборки.

Вы можете настроить выполнение этой задачи с точными интервалами времени, необходимыми для вашего конкретного сценария. Также в зависимости от ваших потребностей эта новая задача может иметь более низкий или более высокий приоритет, чем задача GUI.

Если у него более высокий приоритет, то вам гарантируется, что он выполняется точно в указанное вами время, независимо от того, что выполняет задача GUI. У этого есть недостаток, заключающийся в том, что если это процесс, потребляющий процессор, это может повлиять на частоту кадров пользовательского интерфейса.

Если, с другой стороны, выборка не критична по времени, вы можете назначить задаче более низкий приоритет, чем задаче графического интерфейса, чтобы выборка окружающей системы никогда не влияла на частоту кадров пользовательского интерфейса. Задача GUI будет часто переходить в режим ожидания во время рендеринга (например, при ожидании завершения передачи пикселей на основе DMA), поэтому задачам с более низким приоритетом будет разрешено выполняться довольно часто, и поэтому этого достаточно для подавляющего большинства приложений.

Если вы используете подход вторичной задачи, мы рекомендуем вам воспользоваться системой обмена сообщениями между задачами, предоставляемой вашими ОСРВ. Большинство, если не все, RTOS имеют механизм очереди / почты, который позволяет отправлять данные (обычно определяемые пользователем структуры C, массивы байтов или простые целые числа) из одной задачи в другую. Чтобы передать новые данные в задачу GUI, настройте почтовый ящик или очередь сообщений для задачи UI и отправьте данные в задачу GUI с помощью этой системы обмена сообщениями. Затем вы можете из Model::tick опросить почтовый ящик задачи GUI, чтобы проверить, поступили ли какие-либо новые данные. В этом случае прочитайте данные и соответствующим образом обновите пользовательский интерфейс.

# Передача данных в пользовательский интерфейс

Независимо от того, используете ли вы [выборку из задачи GUI](https://support.touchgfx.com/docs/development/ui-development/touchgfx-engine-features/backend-communication#sampling-from-gui-task) или [выборку из вторичной задачи](https://support.touchgfx.com/docs/development/ui-development/touchgfx-engine-features/backend-communication#sampling-from-secondary-task), Model::tick функция - это место, где задаче GUI становится известно о новых данных, которые будут показаны в пользовательском интерфейсе. Напомним, что класс Model не только выполняет функции интерфейса к вашей окружающей системе, но и отвечает за хранение данных о состоянии, поэтому могут существовать некоторые переменные состояния, которые также нуждаются в обновлении.

Давайте рассмотрим простой пример, когда к системе подключен датчик температуры, и что текущая температура должна отображаться в пользовательском интерфейсе. При подготовке мы расширим класс Model для поддержки этого:

##### Модель.hpp

*class* Model  
{  
*public*:  
 *// Function that allow your Presenters to read current temperature.*  
 *int* getCurrentTemperature() *const* { *return* currentTemperature; }  
  
 *// Called automatically by framework every tick.*  
 *void* tick();  
 ...  
*private*:  
 *// Variable storing last received temperature;*  
 *int* currentTemperature;  
 ...  
};

Благодаря вышесказанному ваш Presenters (ведущий) может запрашивать модель о текущей температуре, позволяя ведущему устанавливать это значение в пользовательском интерфейсе (View) при переключении на экран, отображающий температуру. Что нам нужно сделать сейчас, так это иметь возможность снова обновить пользовательский интерфейс при получении новой информации о температуре. Для этого мы используем тот факт, что Модель имеет указатель на ваш текущий активный презентатор. Тип этого указателя — это интерфейс (ModelListener), который вы можете изменить, чтобы отразить соответствующие события, относящиеся к конкретному приложению:

##### ModelListener.hpp

*class* ModelListener  
{  
*public*:  
 *// Call this function to notify that temperature has changed.*  
 *// Per default, use an empty implementation so that only those*  
 *// presenters interested in this specific event need to*  
 *// override this function.*  
 *virtual* *void* notifyTemperatureChanged(*int* newTemperature) {}  
};

Теперь, когда мы подключили этот интерфейс, осталось выполнить фактическую выборку входящих событий "*новой температуры*" Model::tick

##### Model.cpp

*void* Model::tick()  
{  
 *// Pseudo-code for sampling data*  
 *if* (OS\_Poll(GuiTaskMBox))  
 {  
 *// Here we assume that you have defined a "Message" struct containing type and data,*  
 *// along with some event definitions.*  
 *struct* Message msg = OS\_Read(GuiTaskMBox);  
 *if* (msg.eventType == EVT\_TEMP\_CHANGED)  
 {  
 *// We received information that temperature has changed.*  
 *// First, update Model state variable*  
 currentTemperature = msg.data;  
  
 *// Second, notify the currently active Presenter that temperature has changed.*  
 *// The modelListener pointer points to the currently active Presenter.*  
 *if* (modelListener != 0)  
 {  
 modelListener->notifyTemperatureChanged(currentTemperature);  
 }  
 }  
 }  
}

Описанный выше подход обеспечивает две вещи:

1. currentTemperature переменная всегда актуальна, так что ваш ведущий может в любое время получить текущую температуру.
2. Presenter немедленно получает уведомление об изменениях температуры и может предпринять соответствующие действия.

Одним из преимуществ шаблона MVP является то, что вы достигаете раздельной обработки уведомлений в зависимости от того, на каком экране вы находитесь в данный момент. Предположим, например, что событие изменения температуры происходит при отображении какого-либо меню настроек *(например, активен MainMenuPresenter / MainMenuView)*, где текущая температура не имеет значения.

Поскольку notifyTemperatureChanged функция имеет пустую реализацию по умолчанию, это уведомление просто игнорируется MainMenuPresenter. С другой стороны, если у вас есть TemperatureControlPresenter, вы можете в этом презентаторе переопределить notifyTemperatureChanged функцию и сообщить представлению, что оно должно отображать обновленную температуру:

##### TemperatureControlPresenter.hpp

*class* TemperatureControlPresenter : *public* ModelListener  
{  
*public*:  
 *// override the empty function.*  
 *virtual* *void* notifyTemperatureChanged(*int* newTemperature) {  
 view.setTemp(newTemperature);  
 }  
};

Класс View TemperatureControlView, конечно, должен реализовывать setTemp метод.

# Передача данных из пользовательского интерфейса в окружающую систему

Обратное направление, в котором данные / события передаются из пользовательского интерфейса в окружающую систему, выполняется через модель почти таким же образом. Продолжая предыдущий пример, если нам нужно добавить возможность настройки новой целевой температуры, мы бы добавили в модель следующее:

##### Модель.hpp

*void* setNewTargetTemperature(*int* newTargetTemp)  
{  
 *// Pseudo-code for sending an event to a task responsible for controlling temperature.*  
 *struct* Message msg;  
 msg.eventType = EVT\_SET\_TARGET\_TEMP;  
 msg.data = newTargetTemp;  
 OS\_Send(SystemTaskMBox, &msg);  
}

В случае, если пользователь устанавливает новую целевую температуру в пользовательском интерфейсе, представление может сообщить об этом презентатору, который содержит указатель на объект модели и, следовательно, может вызвать setNewTargetTemperature функцию.

# Примеры

Следующие примеры являются демонстрациями для конкретной платы (BSD), однако большая часть продемонстрированного кода может быть повторно использована для других демонстрационных плат и пользовательского оборудования. Для этих примеров мы создаем задачи и очереди в STM32CubeMX. Затем мы заполняем сгенерированные задачи и реализуем пример пользовательского кода в main\_user.c. В примерах используется библиотека STM32CubeMX BSP librarie для управления светодиодами, пользовательскими кнопками и другими периферийными устройствами в наборах проверки STM32.

## Из задачи GUI

Пример приложения, BSD, можно найти в последней версии TouchGFX Designer в разделе Демонстрации -> Демонстрация для конкретной платы -> Индикатор управления STM32F46G Discovery kit с графическим интерфейсом.

Приложение демонстрирует, как выполнять выборку кнопки и управлять светодиодом. Класс Model выполняет выборку кнопки и обновляет светодиод в соответствии с состоянием приложения.

## Из другой задачи

Пример приложения, BSD, можно найти в последней версии TouchGFX Designer в разделе Демонстрации -> Демонстрация для конкретной платы -> Задача аналогового сэмплера для оценочной платы STM32H7B3I.

Приложение демонстрирует, как выполнять выборку аналогового ввода в отдельном потоке. В примере используется архитектура MVP для передачи аналогового значения в представление.

Пример приложения, BSD, можно найти в последней версии TouchGFX Designer в разделе Демонстрации -> Демонстрация для конкретной платы -> STM32F46G Discovery kit Intertask Communication.

Приложение демонстрирует межзадачную связь и распространение в пользовательский интерфейс и из него. Используйте это как источник вдохновения для вашей собственной настройки. В примере осуществляется обмен данными между серверной системой, реализованной в коде C, и графическим интерфейсом TouchGFX на C ++. Пример выполняется на плате STM32F746G-DISCO поверх FreeRTOS.

## Из нескольких задач

Пример приложения, BSD, можно найти в последней версии TouchGFX Designer в разделе Демонстрации -> Демонстрация для конкретной платы -> Демонстрация многоадресной связи STM32F769I Discovery.

Приложение определяет состояние кнопки, передает сообщение через очередь сообщений графического интерфейса пользователя, если кнопка нажата. Это позволяет нам продвигать анимацию в приложении, удерживая кнопку нажатой.

Приложение использует три задачи FreeRTOS. По одной для графического интерфейса пользователя, по одной для каждого периферийного устройства (светодиод и пользовательская кнопка).

## Из задачи и внешней линии прерывания

Пример приложения, BSD, можно найти в последней версии TouchGFX Designer в разделе Демонстрации -> Демонстрация для конкретной платы -> Демонстрация линии внешнего прерывания STM32F769I Discovery.

Приложение было разработано для платы STM32F769I-DISCO и взаимодействует со светодиодом и кнопкой ПОЛЬЗОВАТЕЛЯ, чтобы показать, как интегрировать код на C и аппаратную периферию в ваше приложение TouchGFX.

Это приложение настраивает кнопку в режиме EXTI (строка внешнего прерывания 0). Поведение заключается в получении прерывания при нажатии кнопки, после чего прерывание сбрасывается. Это не допускает такого же поведения, как в GPIO, но вместо этого мы будем выполнять анимацию одним шагом, потому что сообщение отправляется только через очередь сообщений gui всякий раз, когда получено прерывание.

Приложение использует две задачи FreeRTOS. Одна для графического интерфейса, другая для светодиода. (Задача кнопки из [демонстрации нескольких задач](https://support.touchgfx.com/docs/development/ui-development/touchgfx-engine-features/backend-communication#from-multiple-tasks) остается активной в этом приложении, чтобы продемонстрировать, что код периферийного взаимодействия был перенесен в обработчик прерываний).