Стек протоколов BitCloud фирмы Atmel

Часть 2. Взаимодействие между компонентами стека и прикладной программой

Разработка сетевого приложения для ZigBee-сетей имеет свои особенности — в той же степени, как отличается программирование персонального компьютера и микроконтроллера. Это касается и работы сетевых протоколов, и ограниченности ресурсов узлов сети, и деталей реализации каждого конкретного фирменного ZigBee-стека. В этой части публикаций будут рассмотрены особенности применения стека протоколов BitCloud фирмы Atmel для создания сетевых приложений.

Александр Калачев

Создание приложений для любого типа сетей имеет свои особенности, несмотря на то, что обычно низкоуровневые сетевые операции стараются спрятать. Сети ZigBee [1, 2] не являются исключением. В первой статье серии публикаций на данную тему были рассмотрены особенности организации стека протоколов BitCloud фирмы Atmel [3, 4], являющегося одной из предлагаемых производителями беспроводных компонентов реализаций ZigBee-стека. BitCloud свободно предоставляется в виде предварительно скомпилированных объектных файлов и исходных текстов и представляет собой набор API-функций. Аппаратной платформой данного стека являются интегральные модули для поверхностного монтажа, содержащие 8- или 32-битные микроконтроллеры и беспроводные приемопередатчики для диапазонов 700/800/900-МГц или 2,4-ГГц диапазона [5].

Основным подходом при программировании приложений с использованием BitCloud является событийно ориентированное программирование — определяется набор состояний сетевого узла, прописываются реакции приложения на возникающие события: прерывания, работа с периферийными устройствами и др. Важнейшие моменты работы сети — инициализация, обмен данными, маршрутизация, присоединение или выход из сети узлов.

Инициализация сетевых операций

Процедура старта сети в стеке BitCloud (рис. 1) одинакова для всех типов устройств и выполняется асинхронной процедурой ***ZDO_StartNetworkReq()***. После выполнения вызова, компонент стека ZDO извещает приложение о результатах операции в структуре типа ***ZDO_StartNetworkConf_t***, содержащей информацию о статусе операции (***ZDO_STATUS_SUCCESS*** или ***ZDO_FAIL_STATUS***) и о параметрах новой сети, включая сетевые параметры узла [6].

После подключения к сети конечного узла родительский узел получает подтверждение через функцию ***ZDO_MgmtNwkUpdateNotf()*** со статусом ***ZDO_CHILD_JOINED_STATUS (0x92)***. Возвращаются также сетевой и расширенный адрес узла. Такое подтверждение не формируется при присоединении маршрутизатора к сети, т. к. он не имеет определенного родительского узла.

Обмен данными

Обмен данными является основной задачей при формировании сети ZigBee (как, наверное, и любой другой сети). Обмен данными в сетях ZigBee идет между так называемыми конечными точками, которые представляют конкретные приложения на каждом из устройств. Для осуществления обмена устройство должно зарегистрировать как минимум одну конечную точку, используя функцию ***APS_ReqisterEndPoint()*** с аргументами типа ***APS_RegisterEndpointReq_t type***. Аргумент описывает дескриптор конечной точки — идентификатор (1–240), профайл приложения, количество и список поддерживаемых входных/выходных кластеров, а также поле ***APS_DataInd***,

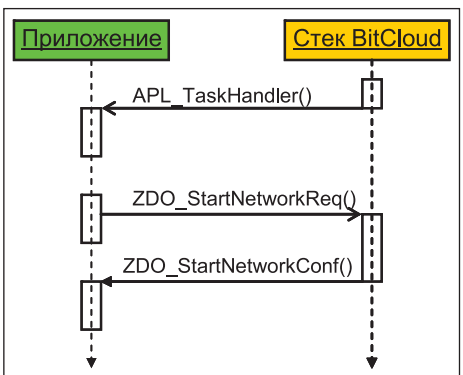

Рис. 1. Диаграмма последовательностей процедуры старта сети в BitCloud

указывающее callback-процедуру, которая будет вызвана, когда для данной конечной точки придут данные.

Приведенный ниже пример иллюстрирует объявление конечной точки 1 с профайлом 1 без ограничения на поддерживаемые кластеры:

```
// определение дескриптора конечной точки:
static SimpleDescriptor_t simpleDescriptor = {1, 1, 1, 1, 0, 0,
NULL, 0, NULL};
// переменная для зарегистрированной конечной точки:
static APS_RegisterEndpointReq_t endpointParams;
// установка свойств конечной точки:
endpointParams.simpleDescriptor = &simpleDescriptor;
endpointParams.APS_DataInd = APS_DataIndication;
// регистрация конечной точки:
APS_RegisterEndpointReq(&endpointParams);
```

Для передачи данных приложение должно сделать запрос типа ***APS_DataReq_t***, который определит полезную нагрузку передаваемого сетевого пакета (Application Specific Data Unit, ASDU) и параметры передачи. Определить callback-процедуру (поле — ***APS_DataConf***), которая будет вызвана для информирования приложения о результатах передачи. Поскольку стек запрашивает ASDU как непрерывный блок данных в памяти с заданными характеристиками, приложение такую структуру должно сформировать.

Максимально допустимый объем передаваемых в одном пакете данных равен 84 байт для нешифрованных передач и 53 байт, если используется стандартный механизм обеспечения безопасности.

Пример формирования корректной структуры ASDU [6]:

```
// Дескриптор буфера сообщений
BEGIN_PACK
typedef struct
{
uint8_t header[APS_ASDU_OFFSET]; // заголовок
uint8_t data[APP_ASDU_SIZE]; // данные приложения
uint8_t footer[APS_AFFIX_LENGTH-APS_ASDU_OFFSET];
// «хвост» пакета
} PACK AppMessageBuffer_t;
END_PACK
static AppMessageBuffer_t appMessageBuffer;
// Буфер сообщений
APS_DataReq_t dataReq; // Запрос на передачу данных:
dataReq.asdu = appMessageBuffer.data;
dataReq.asduLength = sizeof(appMessageBuffer.data);
```

Если используется прямая схема адресации, узел должен знать все параметры приемника — адрес, профайл приложения, конечную точку приложения, поддерживаемые кластеры. При косвенной маршрутизации эти параметры определяются при выполнении процедуры установления связи.

После формирования запроса на передачу данных и установку параметров приложение может передавать данные на уровень APS для трансляции их непосредственно в эфир (рис. 2) — при помощи функции ***APS_DataReq()***.

Если приложение пересылает данные источнику впервые, автоматически сгенерируется запрос на маршрутизацию и будет найден наиболее приемлемый путь до приемника. Более того, этот путь с течением времени обновляется в зависимости от изменений в текущей топологии сети и изменения качества связи. Приложением может быть задано время жизни пакета (количество хопов) и ограничен радиус распространения.

Приложение может запросить подтверждение приема данных на уровне приложения (т. н. APS ACK), устанавливая флаг ***acknowledgedTransmission*** в поле ***txOptions*** запроса данных в 1. В этом случае приложение будет уведомлено об успешном получении пакета ***APS_SUCCESS_STATUS*** через зарегистрированную функцию обратного вызова. Если в течение времени ***CS_ACK_TIMEOUT*** ответа не будет, приложение получит статус доставки пакета ***APS_NO_ACK_STATUS***. Если получение подтверждения отключено и все параметры пакета выставлены корректно, приложение получит подтверждение ***APS_SUCCESS_STATUS*** сразу после отправки пакета в эфир. Функция станет доступной для следующего вызова только после выполнения подтверждающего callback-вызова.

Помимо адресных передач, приложение может передавать данные всем узлам сети или выбранной группе узлов с заданными свойствами:

- передача всем узлам сети — ***BROADCAST_ADDR_ALL (or 0xFFFF)***;
- всем узлам с параметром ***rxOnWhenIdle=1*** — ***BROADCAST_ADDR_RX_ON_WHEN_IDLE (or 0xFFFD)***;
- всем маршрутизаторам сети — ***BROADCAST_ADDR_ROUTERS (or 0xFFFC)***.

Широковещательные пакеты отправляются без подтверждения (***acknowledgedTransmission=0 в поле txOptions***).

На сетевом уровне широковещание организовано следующим образом: после вызова функции ***APS_DataReq()*** с широковещательным адресом в пакете передатчик транслирует пакет в эфир 3 раза. Каждый из узлов после получения копии пакета (обрабатывается только одна копия, остальные игнорируются) уменьшает радиус распространения на 1, и, если он более 0, в свою очередь передает копию в эфир 3 раза. Процедура повторяется до исчерпания радиуса распространения пакета.

Кроме собственно широковещания, приложение может запросить передачу пакета всем конечным точкам выбранного узла (поле ***dstEndpoint= APS_BROADCAST_ENDPOINT (or 0xFF)***). Если радиус распространения установлен в 0, то все узлы, принадлежащие данному типу, охватываются транзакцией. Для процедуры приема данных приложение должно зарегистрировать как минимум одну конечную точку.

После приема узлом пакета данных стек протоколов проверяет совпадение номеров конечных точек, указанных в пакете и зарегистрированных приложением. Если совпадения есть, соответствующие функции, как указывалось в поле ***APS_DataInd*** запроса регистрации конечной точки ***APS_RegisterEndpointReq_t***, будут выполнены с аргументом типа ***APS_DataInd_t***. Аргумент содержит данные приложения, информацию об источнике, приемнике, конечной точке, профайле и т. п.

Для доставки данных конечным точкам на последнем этапе используется механизм опроса родительского узла. Для пакетов, адресованных дочерним узлам, и широковещательных пакетов родительский узел буферизирует пакет и ожидает запроса от дочернего узла. В активном режиме дочерний узел опрашивает родительский с периодичностью ***CS_INDIRECT_POLL_RATE мс*** (опрос производится на сетевом уровне и прозрачен для приложений). После получения запроса от конечного устройства «родитель» передает буферизированные данные устройству и указывает, есть ли еще данные для него. По окончании приема данных от родительского узла конечное устройство может переходить в спящий режим (рис. 3).

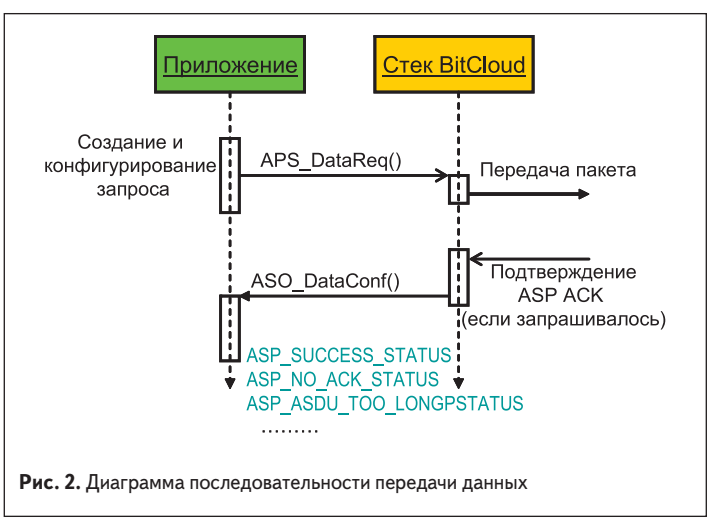

Рис. 2. Диаграмма последовательности передачи данных

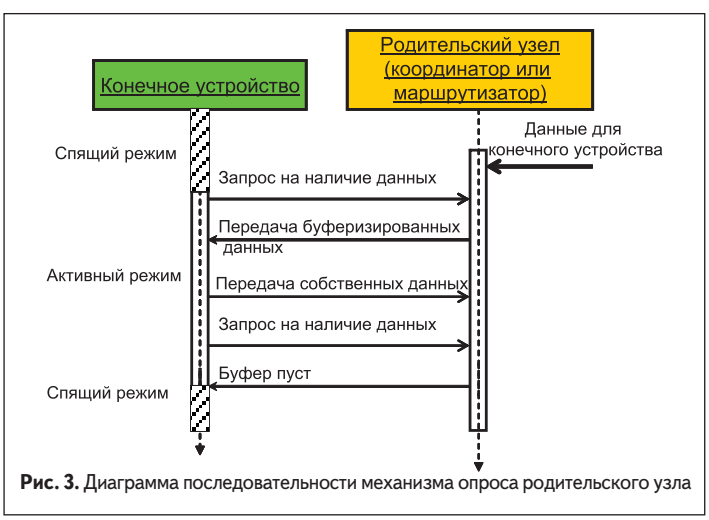

Рис. 3. Диаграмма последовательности механизма опроса родительского узла

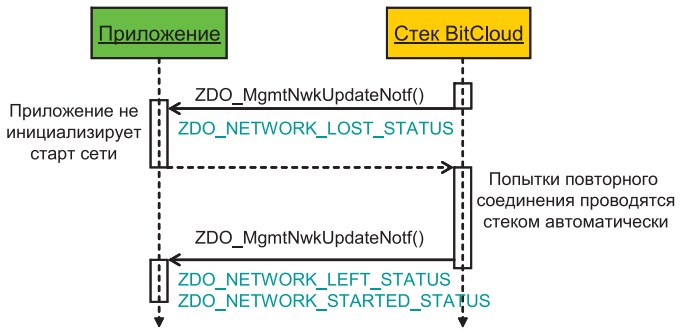

Рис 4. Диаграмма последовательностей потери родительского узла

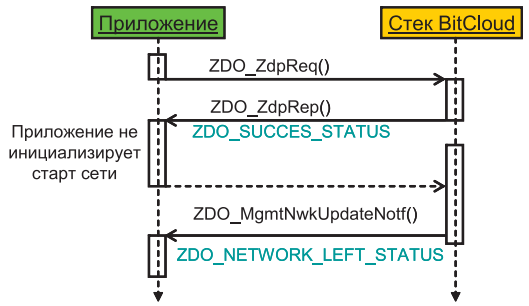

Рис. 5. Диаграмма последовательности выхода узла из сети по требованию приложения

Отслеживание изменений в сети

Функционирование сенсорной сети может происходить в различных условиях — подвижные узлы сети, изменение качества радиосвязи, ограниченные энергетические ресурсы. В этой связи достаточно часто возникают ситуации, при которых один или несколько узлов сети выходят из ее состава. Различают два варианта выхода узла из сети: в результате технических проблем (незапланированный выход) и штатный (согласно работе сетевого приложения или по указанию другого узла сети).

При незапланированном выходе возможна обработка двух типов событий — потеря родительского узла (только для конечных устройств) и потеря дочернего узла (для маршрутизаторов и координатора).

Потеря родительского узла конечным устройством вызывает информационное сообщение сетевого протокола ***ZDO_MgmtNetworkUpdateNotf()*** со статусом ***ZDO_NETWORK_LOST_STATUS*** [6]. Конечное устройство не может соединиться с родительским узлом. При получении этого сообщения приложение может предпринимать определенные действия, однако не должно начинать процедуру присоединения/повторного соединения с сетью. Подключение к сети восстановится автоматически при восстановлении связи с родительским узлом — стек пытается обнаружить родительский узел. Результат передается узлу сообщением ***ZDO_MgmtNwkUpdateNotf()*** со статусом ***ZDO_NETWORK_STARTED_STATUS*** в случае успеха и сетевыми параметрами, отображенными в аргументах функции. Если процедура восстановления подключения к сети вернула статус ***ZDO_NETWORK_LEFT_STATUS***, узел может предпринимать действия по изменению сетевых параметров и инициализации процедуры подключения к сети (рис. 4).

Часто родительскому узлу необходимо регистрировать события выхода дочернего узла из сети, такие события могут быть сгенерированы или ассоциированы только с конечными устройствами. Особенно это важно в связи с тем, что конечные узлы могут иметь длительные периоды нахождения в спящем состоянии и не иметь данных для передачи. С другой стороны, бывают периоды очень активного обмена данными, например при выходе из режима сна.

Таким образом, чтобы убедиться в отсутствии узла в сети, родительский узел должен знать о длительности периода сна дочернего узла (***CS_END_DEVICE_SLEEP_PERIOD***). Предполагается, что при выходе из спящего режима конечное устройство выполняет запрос на отложенные данные для узла. Если в течение интервала ***CS_NWK_END_DEVICE_MAX_FAILURES *(CS_END_DEVICE_SLEEP_PERIOD + CS_INDIRECT_POLL_RATE)*** дочерний узел не выполняет запрос данных, родительский узел полагает, что узел вышел/выпал из сети, и стек генерирует сообщение ***ZDO_MgmtNwkUpdateNotf()*** со статусом ***ZDO_CHILD_REMOVED (0x93)*** с аргументами, содержащими адрес потерянного узла.

Штатный выход из сети производится при помощи процедуры ***ZDO_ZdpReq()*** (рис. 5) [6]:

```
static ZDO_ZdpReq_t zdpLeaveReq;
//Установить соответствующий идентификатор кластера - ID
zdpLeaveReq.reqCluster = MGMT_LEAVE_CLID;
zdpLeaveReq.dstAddrMode = EXT_ADDR_MODE;
zdpLeaveReq.dstExtAddr = 0; // для самого узла адрес=0
zdpLeaveReq.ZDO_ZdpResp = ZDO_ZdpLeaveResp; // callback-функция
//f для самого узла адрес=0
zdpLeaveReq.req.reqPayload.mgmtLeaveReq.deviceAddr = 0;
// определяем, необходимо ли выводить дочерние узлы из сети или нет
zdpLeaveReq.req.reqPayload.mgmtLeaveReq.removeChildren = 0;
//следует ли выполнять процедуру соединения после выхода из сети
zdpLeaveReq.req.reqPayload.mgmtLeaveReq.rejoin = 1;
ZDO_ZdpReq(&zdpLeaveReq); // запрос на выход из сети
```

Если в коде указать адрес удаленного узла, то после вызова ***ZDO_ZdpReq(&zdpLeaveReq)*** он передаст сетевую команду на выход из сети на удаленный узел (рис. 6).

Управление энергопотреблением узлов

В сетях ZigBee уровень энергопотребления устройств достаточно важен, иногда критичен, поскольку не все узлы могут иметь стационарное питание. Стек BitCloud предоставляет ряд API, позволяющих изменять режим работы устройства (активный/спящий), выключать приемопередатчик для уменьшения энергопотребления. Согласно текущему стандарту ZigBee PRO, управление питанием доступно только для конечных устройств — координатор и маршрутизаторы являются всегда активными.

Независимо от своего сетевого статуса, конечное устройство может быть в активном или спящем состоянии. После включения питания узел находится в активном режиме — микропроцессор полностью включен, приемопередатчик готов к приему/передаче данных, приложение может вызывать любую команду BitCloud-стека и получать соответствующие уведомления. В спящем режиме приемопередатчик отключается, микропроцессор переводится в энергосберегающий режим. В BitCloud пробуждение возможно только от самого МПУ и по приходу радиосигнала — т. е. приложение не может инициировать операции приема/передачи данных, взаимодействовать с периферией и т. п.

Чтобы перевести узел в спящий режим, приложение должно вызвать функцию

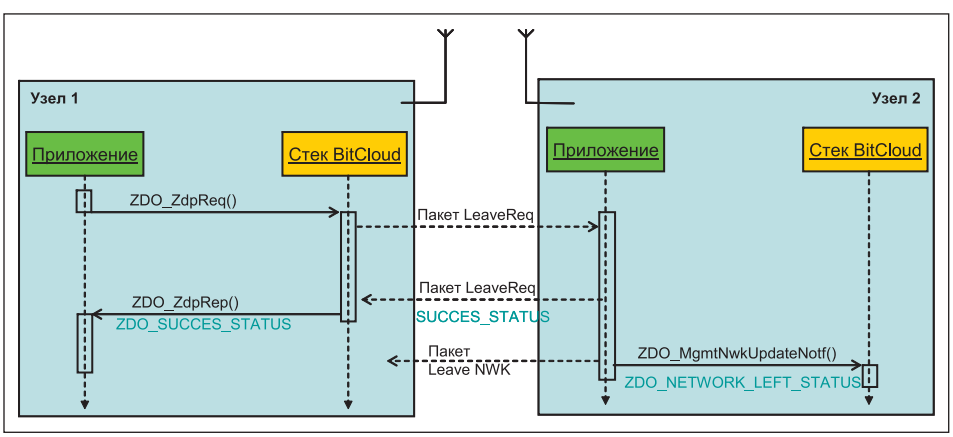

Рис. 6. Диаграмма последовательности вывода удаленного узла из сети по запросу другого узла (узел 1 посылает команду на выход из сети узла 2)

ZDO_SleepReq() с аргументом типа ***ZDO_SleepReq_t***. Зарегистрированный обратный вызов вернет статус операции, и если он равен ***ZDO_SUCCESS_STATUS***, узел перейдет в спящий режим.

Существует два пути пробуждения узла — по расписанию и по прерыванию (IRQ) (рис. 7 и 8 соответственно). При пробуждении по плану узел выйдет из спящего режима через ***CS_END_DEVICE_SLEEP_PERIOD мс***. Приложение будет уведомлено функцией ***ZDO_WakeUpInd***. Параметр ***CS_END_DEVICE_SLEEP_PERIOD*** не может быть изменен во время выполнения.

При пробуждении по прерыванию процессор переводится в активный режим одним из зарегистрированных IRQ-событий, которое будет обработано HAL-уровнем без участия сетевой части стека. Для приведения в активное состояние всего стека протоколов приложение должно вызвать функцию ***ZDO_WakeUpReq()***. После того как стек даст подтверждение ***ZDO_SUCCESS_STATUS***, стек, приемопередатчик и микропроцессор полностью готовы к работе. Если IRQ-событие возникает в спящем режиме и таймер отсчитывает интервал ***CS_END_DEVICE_SLEEP_PERIOD***, то таймер будет остановлен и перезапущен при следующем вызове ***ZDO_SleepReq()***. Если параметр ***CS_END_DEVICE_SLEEP_PERIOD=0***, только IRQ-событие может вывести узел из спящего режима.

В некоторых случаях необходимо сохранить функционирование микропроцессора, а приемопередатчик отключить для уменьшения энергопотребления. Для этого приложение должно вызвать функцию ***ZDO_StopSyncReq()***. Повторный старт радиочасти осуществляется процедурой ***ZDO_StartSyncReq()***. При этом важно учитывать факт, что родительский узел ожидает от конечного узла запроса на получение данных в течение определенного интервала времени и в случае неполучения запроса посчитает узел потерянным или выпавшим из сети.

Структура пользовательских приложений для BitCloud

Структура приложения для BitCloud существенно отличается от структуры обычной программы, скажем для ПК (не для встроенной системы) [6, 7]:

- Каждое приложение определяет свой идентификатор задачи, который содержит данные о коде приложения (включая код, доступный через вложенные функции).
- Каждое приложение определяет некоторое число функций обратного вызова, которые содержат код, выполняющийся при асинхронных запросах. сделанных нижележащими уровнями.
- Приложение определяет ряд функций обратного вызова с известными именами, которые выполняются, когда какое-либо событие обрабатывается стеком.
- Каждое приложение устанавливает свое глобальное состояние, которое становится известным функциям обратного вызова и менеджеру задач (идентификатору задач).

Функция ***main*** встроена в стек протоколов. При инициализации стека управление передается от ***main*** к планировщику задач, который принимает идентификаторы задач в порядке приоритета, вызывая по событиям ***APL_TaskHandler***, который является точкой входа в приложение. При первоначальном вызове идентификатора управление передается от стека протоколов функциям обратного вызова.

Как правило, определяется глобальная переменная, в примере ниже — это ***appState***, которая доступна функциям обратного вызова (callback-функциям) и менеджеру задач. Реально состояние модуля будет представлять собой набор переменных состояния устройства, зависящих от его роли в сети, набора сетевых параметров, состояния сенсоров и пр. В ходе работы, приложение изменяет состояние модуля в зависимости от текущих переменных состояния и результатов работы callback-функций или сообщений стека протоколов.

Разработчик приложения должен в первую очередь описать приложение в терминах конечного автомата. Переход от автомата к коду в итоге даст событийно ориентированный стиль программирования.

Следует также обращать внимание на использование функции планировщика заданий ***SYS_PostTask***. Для примера, приложение прерывает выполнение по причине потери сети, передавая это менеджеру задач. Callback-функция просто изменяет глобальное состояние и возвращается к ZDO-уровню. Такой стиль программирования совместим с кооперативной многозадачностью и позволяет стеку протоколов закончить более приоритетную задачу перед возвратом к прерванному выполнению прикладной задачи.

При разработке приложения рекомендуется следовать системным правилам, связанным с особенностями работы стека BitCloud:

1. Все пользовательские приложения организуются как набор процедур обратного вызова, выполняющихся по завершении запросов к нижележащим уровням стека.
2. Пользовательское приложение ответственно за определение процедур, отвечающих за возникающие системные события.
3. Все пользовательские функции обратного вызова должны выполняться не дольше 10 мс.
4. Функции обратного вызова исполняются с приоритетом вызывающего их уровня (или того уровня, к которому они принадлежат).
5. Менеджер прикладных задач выполняется только в том случае, если нет более приоритетных задач.
6. Менеджер прикладных задач должен выполняться менее 50 мс.
7. Длительность критической секции не должна превышать 50 мкс.
8. Структуры данных должны передаваться в функции через указатели.
9. Там, где возможно, следует использовать глобальные переменные.
10. Следует воздерживаться от рекурсивных функций. Рекомендуемая вложенность вызовов пользовательских функций не более 10 (для функций, имеющих не более двух указателей в качестве параметров).
11. Запрещено динамическое распределение памяти, программист должен воздержаться от употребления стандартных функций C: malloc(), calloc(), realloc().
12. Аппаратные ресурсы, используемые стеком протоколов, не должны быть задействованы пользовательским приложением [7].

Код приложения может быть распределен между несколькими C-файлами или помещен

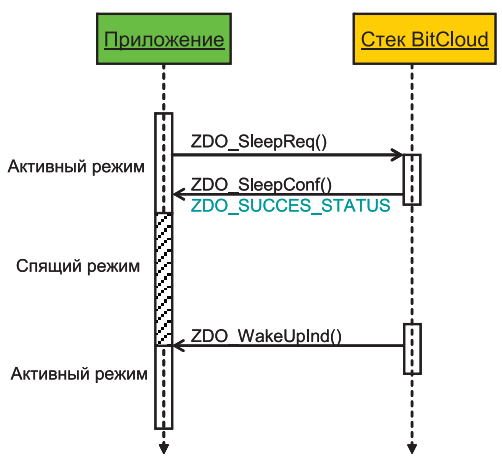

Рис. 7. Процесс запланированного пробуждения узла

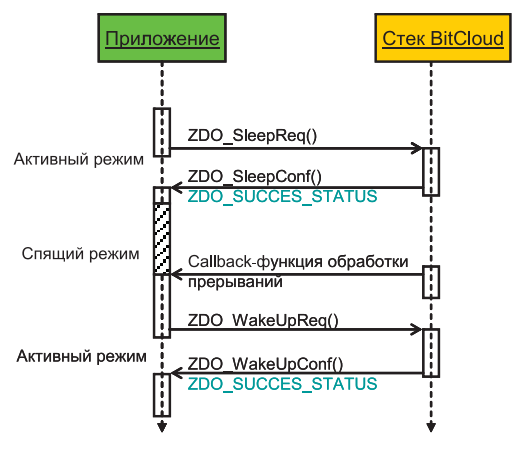

Рис. 8. Пробуждение узла по возникновении прерывания

в один файл. Схематично код типичного приложения выглядит следующим образом [6]:

```
/************************************************
Подключение необходимых компонент стека
************************************************/
...
#include<taskManager.h>
#include<zdo.h>
#include<configServer.h>
#include<aps.h>
/************************************************
Прототипы функций
************************************************/
...
/************************************************
Глобальные переменные
************************************************/
AppState_t appState = APP_INITING_STATE;
...
/************************************************
Реализация приложения
************************************************/
/************************************************
Менеджер задач приложения - Application task handler
************************************************/
void APL_TaskHandler()
{
switch (appState)
{
case APP_IN_NETWORK_STATE:
...
break;
case APP_INITING_STATE: //node has initial state
...
break;
case APP_STARTING_NETWORK_STATE:
...
break;
}
}
/************************************************
Callback-функции подтверждения
************************************************/
static void ZDO_StartNetworkConf(ZDO_
StartNetworkConf_t *confirmInfo)
{
...
if (ZDO_SUCCESS_STATUS == confirmInfo->status)
{
appState = APP_IN_NETWORK_STATE;
...
SYS_PostTask(APL_TASK_ID);
}
}
void ZDP_LeaveResp(ZDP_ResponseData_t *zdpRsp)
{
...
}
/************************************************
Callback-функции индикации состояния
************************************************/
void ZDO_WakeUp_Ind(void)
{
...
if (APP_IN_NETWORK_STATE == appState)
{
appState = APP_STARTING_NETWORK_STATE;
...
SYS_PostTask(APL_TASK_ID);
}
}
void ZDO_MgmtNwkUpdateNotf(ZDO_
MgmtNwkUpdateNotf_t *nwkParams)
{
...
}
```

Заключение

На основе вышесказанного можно сформировать общие требования к этапам разработки сетевого приложения, позволяющие формализовать процесс его разработки, документирования и модернизации.

Основной подход к проектированию приложений с использованием BitCloud — сверху вниз:

- определяются принципы функционирования сетевого приложения в целом;
- строится логическая топология сети, определяются ее размеры и состав узлов;
- определяются параметры узлов в зависимости от их типа;
- для каждого типа узлов строятся диаграммы состояний (если есть необходимость, возможно деление и на группы по функциональному признаку);
- для каждого состояния узла строится диаграмма последовательностей, описывающая взаимодействие приложений с компонентами стека, определяются необходимые callback-функции подтверждения, индикации состояния;
- строятся диаграммы работы callback-функций в зависимости от текущего состояния узла;
- для каждого типа узлов проводится формализация алгоритмов и диаграмм работы на языке C, с учетом правил построения приложений, действующих для стека BitCloud.

Литература

1. www.zigbee.org/Markets/ZigBeeSmartEnergy/Specification.aspx.
2. http://zigbee.org/Markets/ZigBeeRF4CE/download.aspx.
3. www.atmel.com/dyn/products/tools_card.asp?tool_id=4495&family_id=676.
4. www.atmel.com/dyn/products/tools.asp?family_id=676.
5. www.atmel.com/products/zigbee/zigbit_modules.asp?family_id=676.
6. www.atmel.com/dyn/resources/prod_documents/doc8199.pdf.
7. www.atmel.com/dyn/resources/prod_documents/8200.pdf.