CRONYX

Tau32-PCI
DDK

Комплект для разработки драйверов устройства (DDK)
Описание и руководства разработчика (пользователя)

 Версия 1.6, Июль 2006

Содержание

Пожелания и замечания об этом руководстве и работе программного обеспечения, пожалуйста, отправляйте на адрес info@cronyx.ru

Введение

Комплект для разработки драйверов адаптера Tau32 (Driver Development Kit, далее DDK) предназначен для создания драйвера (драйверов) адаптера Tau32, и позволяет полноценно использовать аппаратные возможности устройства в большинстве современных операционных систем.

DDK представляет собой набор функций, констант и структур данных для управления аппаратной частью устройства.

Данный документ является руководством и предназначен для предоставления информации, необходимой для полноценного и надлежащего использования DDK.

Необходимые знания и навыки

Разработчику драйвера (пользователю DDK) необходимо владеть следующим набором знаний:

- Алгоритмические языки программирования C и C++;
- Общие принципы разработки драйверов для PCI-устройств;
- Опыт разработки драйверов устройств для целевой операционной системы;
- Принципы функционирования систем передачи данных на базе каналов E1/ИКМ-30;
- ITU-T рекомендации G.703, G.704, G.706 и G.732;
- Принципы последовательной синхронной передачи данных, основы протокола HDLC;
- Основы сигнализации по выделенным каналам (Channel-Associated Signaling, CAS);

Для более точного описания различных ситуаций взаимодействия с операционной системой, в руководстве приводятся ссылки на документацию DDK ОС Microsoft Windows (2000/XP/2003). Поэтому может быть полезным наличие соответствующей документации и опыт разработки драйверов к ОС Windows 2000/XP/2003.

При описании функций DDK приведены примеры кода для ОС Linux 2.6.x и ОС Windows Server 2003. Поэтому для понимания примеров разработчик драйвера (пользователь DDK), должен быть квалифицирован для разработки драйверов как минимум к одной из этих ОС.

Структура руководства

Руководство состоит из двух основных частей описывающих архитектуру DDK и детального описания (справочника) каждой функции и структуры.

Точное и подробное описание всех возможностей и режимов работы, для удобства приводится во второй части, вместе с описанием соответствующих элементов

DDK. Однако при работе с руководством настоятельно рекомендуется перейти к «Справочнику» только после изучения раздела «Архитектура».

Устройство адаптера

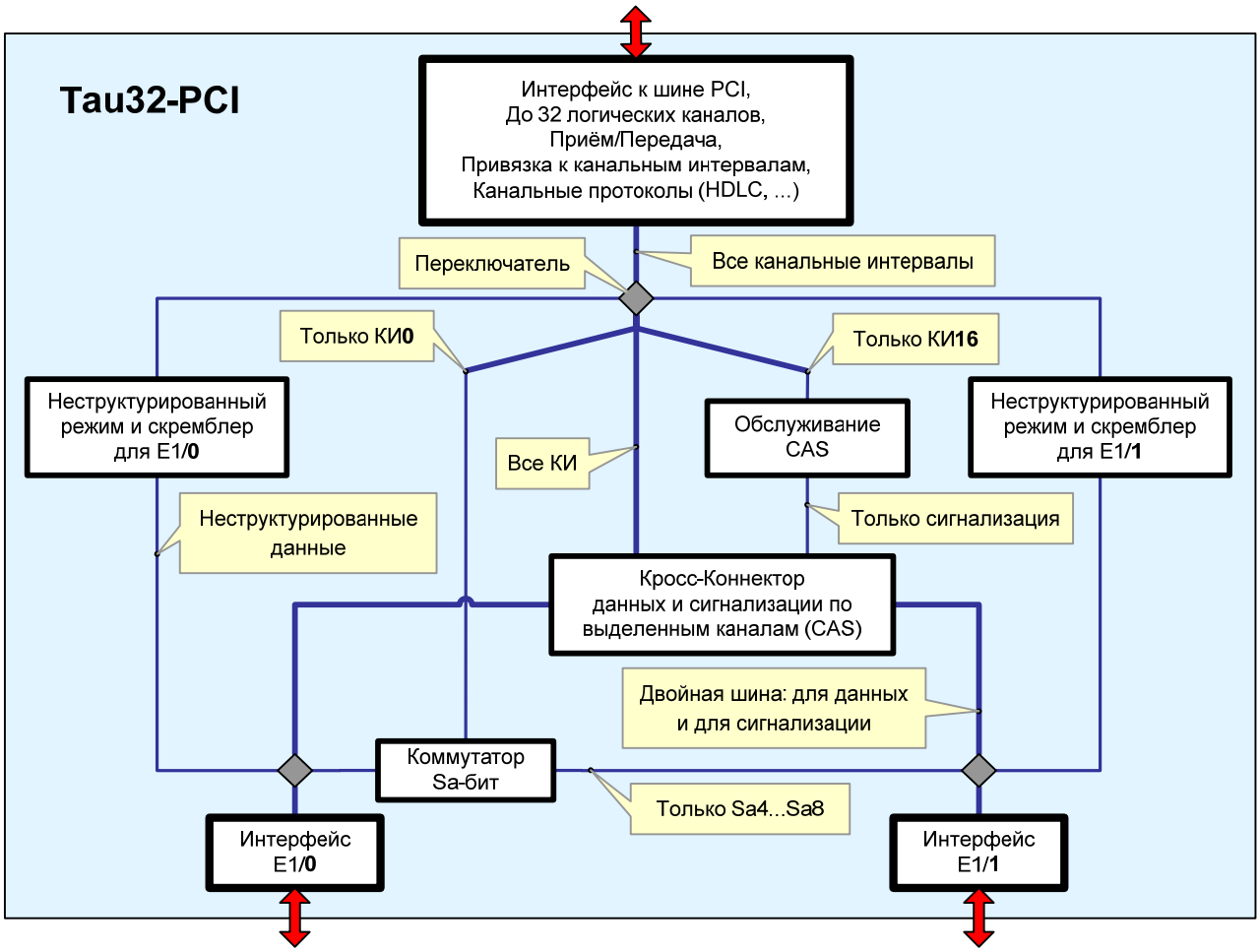
Адаптер Tau32 представляет собой универсальное PCI-устройство с двумя (или одним для модели «Lite») линейными интерфейсами E1, позволяющее решать самые разнообразные, в том числе уникальные задачи. Tau32 может быть незаменим при разработке, отладке и контроле оборудования и приложений, использующих каналы E1/ИКМ-30.

Аппаратная часть адаптера имеет уникальный набор средств и возможностей, полностью поддерживаемых и доступных посредством DDK.

Функционально структуру адаптера Tau32 можно разделить на шесть основных частей:

- Приёмопередатчик, обслуживающий одновременно до 32 логических каналов, сформированных из канальных интервалов E1;
- Интерфейсы линий E1, в зависимости от модели адаптера, может устанавливаться два, либо один интерфейс;
- Неблокируемый кросс-коннектор, осуществляющий произвольную кросс-коммутацию канальных интервалов между тремя «точками»: приёмопередатчиком и двумя интерфейсами E1;
- Отключаемый формирователь данных сигнализации по выделенным каналам (CAS) и независимый кросс-коннектор ниблов сигналинга, позволяющий задействовать $16^{ый}$ канальный интервал приёмопередатчика для обмена данными сигнализации CAS. Блок обслуживания CAS отсутствует в модели Tau32-Lite, поскольку его функциональность становится избыточной (и незатребованной) в случае с одним интерфейсом E1;
- Отключаемый формирователь и коммутатор Sa-битов (Sa4…Sa8), позволяющий задействовать $0^{ой}$ канальный интервал приёмопередатчика под обмен Sa-битами, и одновременно «пробрасывать» Sa-биты между тремя точками: $0^{ым}$ канальным интервалом приёмопередатчика и интерфейсами E1;
- Отключаемые модули поддержки неструктурированного режима и скремблера для каждого из интерфейсов E1;
- Управляемый цифровой генератор тактовой частоты с точностью $\pm 0{,}233\times10^{-3}$ ppm ($\pm 0{,}5\times10^{-3}$ Hz или $\pm 2^{-32}$ опорной частоты);
- Счетчики тактовых импульсов опорного генератора и текущего (выбранного) источника синхронизации;
- Блок формирования единой внутренней синхронизации для обработки данных, в качестве первоисточника задается один из узлов адаптера: внутренний опорный генератор, внутренний управляемый генератор, один из приёмников интерфейсов E1;

Функциональная схема адаптера



Приёмопередатчик реализован на контроллере «Infineon MUNICH32X PEB/PEF 20321», и обеспечивает обмен (приём/передачу) информацией между «логическими каналами» и оперативной памятью компьютера через шину PCI 2.1. Логические каналы приёмопередатчика формируются из задаваемых канальных интервалов E1. Каждый канальный интервал может быть «подключен» только к одному логическому каналу. Это позволяет получить до 32 логических каналов. Обмен данными в логическом канале происходит синхронно, со скоростью зависящей от количества подключенных канальных интервалов (опционально несимметричной). Для каждого логического канала задается канальный протокол (HDLC, прозрачный режим, и т.д.), своя очередь приёма и передачи, каждый канал может запускаться и останавливаться независимо от других.

Отключаемый блок обслуживания сигнализации CAS позволяет организовать обработку сигнализации по выделенным каналам. Блок обслуживания CAS обменивается данными с приёмопередатчиком через 16ый канальный интервал, а с другой стороны взаимодействует с кросс-коннектором по отдельной, параллельной шине данных. Блок обслуживания CAS отсутствует в модели Tau32-Lite, поскольку его функциональность становится избыточной и незатребованной в случае с одним интерфейсом E1.

Неблокируемый кросс-коннектор позволяет для каждого «исходящего» канального интервала задавать «входящий» источник. Это позволяет произвольным образом коммутировать и переставлять канальные интервалы, организовывать цифровые шлейфы, и т.д. Обновление матрицы кросс-коммутации происходит на границе фрейма E1.

Независимый неблокируемый кросс-коннектор сигнализации CAS, который функционирует аналогично основному кросс-коннектору канальных интервалов, но имеет независимую таблицу коммутации и обрабатывает ABCD-ниблы CAS.

Совместное использование блока обслуживания CAS, кросс-коннектора канальных интервалов и кросс-коннектора CAS, одновременно с включением обмена сигнализацией на интерфейсах E1, позволяет организовать полноценную произвольную, трех точечную кросс-коммутацию с поддержкой DACS.

Отключаемый коммутатор Sa-бит позволяет организовать приём/передачу значений Sa-битов и обмен ими между интерфейсами E1. Коммутатор Sa-бит обменивается данными с приёмопередатчиком через $0^{ой}$ канальный интервал, а с другой стороны взаимодействует непосредственно с интерфейсами E1.

Отключаемые модули поддержки неструктурированного режима для каждого интерфейса E1 симметричны, и управляются независимо. Под «неструктурированным» режимом понимается отсутствие в потоке данных E1 кадровой (фреймовой) структуры согласно рекомендации ITU-T G.704, при этом данные передаются синхронным битовым потоком 2048 Кбит/сек, а границы канальных интервалов неопределенны. Соответственно теряет смысл работа кросс-коннектора, блока обслуживания CAS и коммутатора Sa-бит.

При включении неструктурированного режима со скоростью 2048 Кбит/сек (без занижения скорости, см. ниже), на одном из интерфейсов E1, занимаются все канальные интервалы приёмопередатчика. Обмен данными с другим интерфейсом E1, и прочими функциональными блоками адаптера, становится невозможным. Соответственно в приёмопередатчике должен быть активирован один логический канал для всех канальных интервалов.

Для совместимости с другим оборудованием производства «Кроникс», адаптером поддерживается режимы использования скремблера, и кратного занижения скорости (в 2, 4, 8, 16 и 32 раза). Физическая скорость передачи данных при этом всегда составляет 2048 Кбит в секунду, изменяется лишь скорость передачи полезной нагрузки.

В неструктурированных режимах с занижением скорости задействуются не все 32 канальных интервала приёмопередатчика. Управляющим модулям задаются канальные интервалы, которые следует задействовать, и их количество должно соответствовать выбранной скорости. Указанные канальные интервалы отключаются от кросс-коннектора и используются только модулями неструктурированного режима.

При включении неструктурированного режима, шина данных соответствующего интерфейса E1 переключается от кросс-коннектора к управляющему модулю.

Обмен Sa-битами и сигнализацией CAS также теряет смысл, эти данные будут игнорироваться интерфейсом E1.

Канальные интервалы, не задействованные неструктурированным режимом интерфейса E1, могут использоваться произвольно. В том числе для взаимодействия с другим интерфейсом E1, как в структурированном, так и в неструктурированном режимах.

Интерфейсы E1 реализованы на контроллерах «Dallas DS21554». Со стороны DDK обеспечивается поддержка практически всех возможностей этих контроллеров при работе в режиме E1/ИКМ-30.

Дополнительно для интерфейсов E1 в DDK реализовано три режима низкоуровневого побайтового обмена на базе FIFO: режим обмена данными сигнализации CAS, и два режима обмена битами Sa4…Sa8, Ra и опционально Si (так называемого FAS-режима). При задействовании этих режимов темп генерации аппаратных прерываний резко возрастает (см. «Темп генерации аппаратных прерываний»).

Приёмопередатчик, кросс-коннектор, и другие функциональные узлы адаптера всегда синхронизированы от одного источника. В качестве такого источника может использоваться приёмный тракт одного из интерфейсов E1, либо внутренний генератор адаптера. В случае потери синхросигнала от внешнего источника, адаптер всегда автоматически переходит на работу от внутреннего генератора.

Порядок битов в канальных интервалах

В рекомендации ITU-T G.704 биты канальных интервалов и других структур нумеруются в порядке их передачи по физической линии. При этом во многочисленных схемах биты с меньшими номерами изображаются слева.

Если отобразить этот порядок на нумерацию бит в байте, то соответственно младшие биты должны передаваться первыми. Однако по общепринятым соглашениям о передачи данных в сетевых средах, первыми передаются единицы информации, имеющие большую значимость. И соответственно при передаче октета (байта) по сети, первыми передаются старшие (наиболее значащие) биты.

Такая «двойственность» порождает хорошо известную проблему «переворота бит» при параллельной (не побитовой) обработке данных передаваемых в последовательных средах.

Поэтому в адаптере Tau32 реализована возможность «переворота» порядка битов в байтах, при обмене данными через приёмопередатчик. Управляемый «переворот» байтов производится кросс-коннектором при взаимодействии с приёмопередатчиком.

Для каждого из 32 канальных интервалов приёмопередатчика, существует независимый битовый флаг, управляющий «переворотом» битов:

- Если управляющий бит равен «**0**», то для соответствующего канального интервала первый бит будет соответствовать младшему биту компьютерного байта (Intel order);

- Если управляющий бит равен «**1**», то для соответствующего канального интервала первый бит будет соответствовать старшему биту компьютерного байта (Network and Motorola order);

Все вместе, управляющие биты образуют 32-битную управляющую маску, которая дополняет конфигурацию кросс-коннектора.

Стоит отметить, что в протоколе HDLC первыми передаются младшие биты компьютерного байта, а в «телефонных» приложениях первыми обычно передаются старшие биты.

Буферные задержки на пути данных

При прохождении данных через функциональные компоненты адаптера, как при приёме, так и передаче, возникают различные задержки. Прежде всего, задержки обусловлены обработкой информации и её преобразованием из последовательной формы в параллельную, и наоборот. Для оценки суммарной задержки рекомендуется пользоваться следующей схемой:

- Интерфейсы E1 могут буферизировать (задерживать) данные до 62,5 μs с целью восстановления частоты и подавления фазового дрожания (jitter attenuator). Задержка может вноситься интерфейсом E1 либо только при приёме, либо только при передаче, в зависимости от режима работы интерфейса и выбора источника синхронизации;
- Интерфейсы E1 могут буферизировать (задерживать) до 2 кадров E1 (до 250 μs) для выравнивания по единому источнику синхронизации (elastic store). Задержка может вноситься интерфейсом E1 как при приёме, так и при передаче;
- Кросс-коннектор всегда производит задержку данных на один кадр E1 (125 μs) во всех направлениях;
- Блок обслуживания сигнализации по выделенным каналам (CAS) всегда задерживает данные сигнализации на один сверхцикл CAS (2 ms);
- В режимах низкоуровневого обмена, между взаимодействием с FIFO-буферами DDK и приёмом/передачей информации, всегда происходит задержка, равная периоду генерации прерываний в соответствующем режиме (см. «Темп генерации аппаратных прерываний»);
- Приёмопередатчик может производить буферизацию до 256 битов данных для каждого из логических каналов, как при приёме, так и при передаче информации;
- Интерфейс шины PCI (FIFO контроллера DMA) может накапливать до 64 машинных слов (256 байт), как в сторону приёмопередатчика, как и в сторону основной оперативной памяти;

Описание DDK

Архитектура DDK ориентирована на высокопроизводительное асинхронное выполнение запросов, включая приём/передачу данных и настройку (конфигурирование) функциональных частей адаптера. Бóльшая часть работы при этом выполняется обработчиком аппаратных прерываний, асинхронно по отношению к вызову функций DDK. Вызывающий (клиентский) DDK код будет получать уведомления (callbacks) о результатах выполнения асинхронных запросов. Синхронно выполняются лишь те функции, выполнение которых не связанно с обработкой прерываний, и не требует каких-либо пауз (задержек).

Особо стоит отметить, что DDK не производит копирование принимаемых и передаваемых данных во внутренние структуры, весь обмен происходит непосредственно через буфера, определяемые пользователем, в режиме Bus-Master шины PCI. Исключения составляют дополнительные режимы низкоуровневого побайтного обмена на базе FIFO.

Все области (участки) оперативной памяти, подготавливаемые вами для DDK, должны быть доступны из программного кода производящего обработку (обслуживание) аппаратных прерываний. В большинстве операционных систем это означает, что эти участки памяти должны быть выделены из резидентной (не выгружаемой в файл подкачки) зоны.

Возможности

В текущей реализации DDK поддерживает следующие основные возможности:

- Инициализацию и диагностику аппаратной части, построение служебных структур DDK;
- Обработку аппаратных прерываний адаптера с возможностью разделения линии IRQ;
- Приём/передачу информации, в режиме обслуживания очереди запросов;
- Привязку канальных интервалов к каждому логическому каналу приёма/передачи;
- Выбор канального протокола (HDLC, «Прозрачный режим», и др.) для каждого логического канала;
- Управление кросс-коннекторами канальных интервалов и сигнализации CAS;
- Выбор пониженных скоростей передачи и включение скремблера для неструктурированного режима E1 (совместимо с другим оборудованием «Кроникс»);
- Обслуживание потока сигнализации CAS (два режима);
- Управление коммутатором Sa-битов;
- Обслуживание потока Sa-битов (три режима);

- Конфигурирование интерфейсов E1;
- Уведомление пользователя о завершении или отмене поступивших запросов;
- Уведомление пользователя об ошибках приёма/передачи и управлении;
- Уведомление пользователя об изменении состояния интерфейсов E1;
- Настройка управляемого генератора и доступ к счетчикам тактовых импульсов;

Следуем отметить, что многие режимы работы аппаратной части и DDK взаимоисключающие, и не могут быть задействованы одновременно. Например, при работе интерфейса E1 в неструктурированном режиме не имеет смысла обслуживание сигнализации CAS. Но функции DDK позволяют задействовать такие режимы работы, и это, конечно же, не приводит к повреждению оборудования адаптера или сбоям в работе DDK. В общем случае DDK не проверят комплексную корректность заданных режимов работы, так как многие логические конфликты неизбежны при смене режимов работы оборудования. Однако возможные при этом ошибки данных и цикловой синхронизации будут обнаруживаться и соответствующим образом обрабатываться.

DDK поставляется в виде заголовочных h-файлов на языке C, и бинарных (скомпилированных) библиотек функций. Каждая из библиотек содержит все функции DDK в машинных кодах для соответствующей аппаратной платформы и операционной системы (семейства ОС).

Исходные тексты DDK не предоставляются, в случае необходимости, специа-листы «Кроникс» по запросу могут подготовить версию библиотек для плат-форм, не вошедших в базовый комплект поставки.

Идеология

Внутреннее устройство DDK практически полностью абстрагировано (скрыто) от пользователей, и независимо от операционной системы. DDK не использует механизмы синхронизации, не распределяет оперативную память, а при приёме/передаче данных оперирует только физическими (не виртуальными) адресами. Организация взаимодействия с операционной системой, вопросы синхронизации и разделения ресурсов, целиком ложатся на пользователя DDK.

Большинство функций DDK в качестве первого параметра получают указатель на объект TAU32_Controller. Внутренне устройство этой структуры скрыто, в ней организованы все служебные данные DDK. Однако так как выделение оперативной памяти для объектов TAU32_Controller входит в обязанности вашего кода, DDK определяет (раскрывает) размер этой структуры константой TAU32_ControllerObjectSize.

Взаимодействие с DDK организуется в основном при помощи двух других открытых структур TAU32_UserContext и TAU32_UserRequest. В структуре TAU32_UserContext хранятся данные, доступные из DDK напрямую (статус, счет-

чики ошибок и др.), а также информация необходимая для инициализации внутренних структур DDK и организации взаимодействия с вашим кодом (обратные вызовы). Структура TAU32_UserRequest определяет асинхронные запросы, которые DDK получает от пользователя.

Каждому обслуживаемому адаптеру должны соответствовать свои «собственные» экземпляры TAU32_Controller и TAU32_UserContext. Выделение памяти под все объекты, и заполнение TAU32_UserContext, должно производиться вашим кодом перед вызовом DDK, который в свою очередь инициализирует TAU32_Controller и свяжет его с TAU32_UserContext.

В процессе работы ваш код может создавать любое необходимое количество структур TAU32_UserRequest, заполнять их и отправлять на исполнение в DDK. Именно через выполнение асинхронных запросов осуществляется всё взаимодействие с приёмопередатчиком и логическими каналами. DDK позволяет ставить в очередь на выполнение до 512 запросов, 256 из которых могут быть запросами на приём или передачу.

Обработка запросов

При получении запроса, DDK проверяет его корректность, и либо принимает запрос к исполнению, либо отвергает. Если запрос принят к исполнению, ваш код перестает «владеть» запросом до уведомления о завершении выполнения, либо до его отмены. Любой запрос может быть отменён, однако отмена запроса также может потребовать асинхронного выполнения. Поэтому, при отмене запроса DDK либо подтверждает его немедленную отмену, либо уведомит ваш код обратным вызовом. Кроме этого, при отмене управляющих запросов, DDK не гарантирует транзакционность. Рекомендуется считать, что после отмены управляющего запроса, состояние соответствующего логического канала не определено.

Все поступившие запросы DDK распределяет по трём очередям выполнения: очередь запросов на передачу данных, очередь запросов на приём данных, все остальные (управляющие) запросы. Очереди приёма/передачи раздельны для каждого логического канала, а очередь управляющих запросов общая. Запросы в каждой очереди выполняются последовательно, но все очереди выполняются параллельно. DDK позволяет ставить в очереди в сумме до 512 запросов, 256 из которых могут быть запросами на приём или передачу.

Обратите внимание, что DDK не гарантирует завершение выполнения всех запросов в порядке их поступления. DDK гарантирует, что запросы в каждой внутренней очереди будут выполняться последовательно, и что порядок обратных вызовов будет соответствующим. Но DDK не гарантирует, что порядок обратных вызовов по завершению запросов из разных внутренних очередей будет соответствовать порядку поступления самих запросов во входящую очередь DDK.

В обновленной версии DDK для ОС Windows для удобства использования адреса функций прямых вызовов совмещены с DPC-объектами. Соответственно при обработке прерываний DDK либо непосредственно вызывает обратный вызов по его адресу, либо ставит DPC-объект в очередь.

Синхронизация с аппаратными прерываниями

Все основные функции DDK повторно входимы (reenterable, «реентабельны») но при условии, что рекурсивный вызов делается из функций обратного вызова (callback). Но функции DDK вызванные для какого-либо адаптера, ни в коем случае не должны прерываться обработчиком аппаратных прерываний этого адаптера. Поэтому вызовы всех функции DDK, за небольшим исключением, должны быть защищены от прерывания обработчиком аппаратных прерываний (см. KeSynchronizeExecution() в Windows DDK). Исключения составляют несколько специальных случаев:

- Функция TAU32_Initialize() не нуждается в синхронизации, поскольку до её вызова генерация прерываний невозможна;
- Функция TAU32_EnableInterrupts(), но только в том случае, если до её вызова генерация прерываний была запрещена;
- На однопроцессорных системах функции TAU32_DisableInterrupts() и TAU32_DestructiveHalt() также могут использоваться без синхронизации, поскольку сами сразу же запрещают прерывания. Однако для совместимости с многопроцессорными (SMP, Symmetric Multi Processor) системами использовать особенности этих функций не рекомендуется;

Нарушение этих правил может привести к самым разнообразным сбоям. Для своевременного обнаружения таких ошибок, DDK производит соответствующий контроль в обработчике прерываний. При обнаружении недопустимой ситуации, поток выполнения будет остановлен на контрольной отладочной точке (команда 0xCC / INT3 для IA32).

С другой стороны, все функции DDK, кроме функций инициализации и остановки, могут быть вызваны из обратных вызовов. Так, например, обратный вызов, получивший уведомление о завершении передачи (или приёма), может направить к DDK новый запрос. Поэтому и предоставляемые вашим кодом функции обратных вызовов должны быть готовы к тому, что в ответ на вызов любой функции DDK, они будут вызваны рекурсивно ещё раз.

Инициализация и запуск

При инициализации DDK, получает набор определяющих параметров необходимых для работы DDK:

- Виртуальный адрес участка памяти выделенного для объекта TAU32_Controller. Выделенный участок памяти должен быть физически последовательным (непрерывным), т.е. располагаться по физически строго последовательным адресам оперативной памяти (см. MmAllocateContiguousMemory() в Windows DDK);
- Физический адрес участка памяти выделенного для объекта TAU32_Controller (см. MmGetPhysicalAddress() в Windows DDK);

- Виртуальный адрес регистров ввода/вывода адаптера, отображенных в оперативную память. Отображение должно быть произведено с отключением кэширования, а также обратной, пакетной и комбинированной записи со стороны центрального процессора, или всех процессоров, которые будут вызывать функции DDK (см. MmMapIoSpace() в Windows DDK);
- Опционально: адрес обратного вызова уведомления об ошибках приёма/передачи и управления;
- Опционально: адрес обратного вызова уведомления об изменении статуса адаптера и интерфейсов E1;

Все области (участки) оперативной памяти, подготавливаемые вами для DDK, должны быть доступны для программного кода производящего обработку (обслуживание) аппаратных прерываний. В большинстве операционных систем это означает, что эти участки памяти должны быть выделены из резидентной (не выгружаемой в файл подкачки) зоны.

Все эти параметры должны быть корректно подготовлены вашим кодом во взаимодействии со средствами операционной системы. Параметры необходимо подготовить в соответствующих полях экземпляра структуры TAU32_UserContext, адрес которой и передается в DDK (в функцию TAU32_Initialize()).

Основываясь на этой информации, DDK заполняет объект TAU32_Controller, инициализирует аппаратуру адаптера и производит её базовое тестирование. Если параметры корректны и тестирование аппаратуры не выявило неисправностей, DDK возвращает позитивный результат. В противном случае в полях структуры TAU32_UserContext будет содержаться информация об обнаруженных проблемах, а задействовать другие функции DDK будет нельзя.

Инициализация может занять до 100 ms (и более) времени выполнения, поэтому не следует выполнять инициализацию при запрещенных (на процессоре) аппаратных прерываниях или при захваченных ресурсах (объектах синхронизации). Например, в случае ОС Windows, инициализация должна производиться при IRQL = PASSIVE_LEVEL (см. Windows DDK).

Следует отметить, что DDK не владеет информацией (и не нуждается в ней) об аппаратных ресурсах (например линии IRQ) назначенных адаптеру, и не производит какую-либо соответствующую настройку аппаратной части. Адаптер Tau32 является PCI-устройством, и поэтому эти задачи должны выполняются операционной системой и/или BIOS.

Инициализация не вызывает генерацию адаптером аппаратных прерываний, если только оборудование не имеет кардинальных повреждений. До успешной инициализации никакие другие функции DDK не могут быть использованы. Поэтому инициализация должна выполняться перед подключением (средствами операционной системы) к обработке аппаратных прерываний и до каких-либо других запросов к DDK.

Если инициализация прошла успешно, то в соответствующих полях структуры TAU32_UserContext будет находиться корректная информация об адаптере (модель,

количество интерфейсов и др.). При этом генерация аппаратных прерываний будет запрещена, и DDK будет готов к дальнейшим действиям. Далее вам необходимо активировать (подключить средствами операционной системы) обработчик аппаратных прерываний DDK (функцию TAU32_HandleInterrupt()) и затем разрешить генерацию адаптером аппаратных прерываний функцией TAU32_EnableInterrupts().

После этого DDK и адаптер будут полностью готовы к работе.

Остановка

При остановке DDK запрещает генерацию устройством всех аппаратных прерываний, прерывает обработку запросов, приём и передачу данных, выключает передатчики интерфейсов E1, производит выключение тактовых генераторов, и сброс аппаратной части. После остановки аппаратная часть потребляет минимум электрической мощности, поэтому при необходимости режим остановки может использоваться как альтернатива энергосберегающего режима.

Сразу после начала выполнения остановки никакие другие функции DDK не могут быть вызваны. Остановка разрушает внутренние структуры DDK и делает невозможным использование DDK до момента завершения последующей инициализации. Поэтому, если линия IRQ используется в режиме разделения (совместного использования, что является требованием для PCI-устройств), то перед выполнением остановки необходимо запретить генерацию устройством аппаратных прерываний вызовом функции TAU32_DisableInterrupts(). Затем деактивировать (отключить средствами операционной системы) обработчик аппаратных прерываний DDK (функцию TAU32_HandleInterrupt()), и только после этого производить остановку.

Остановка может быть произведена в любой момент времени, но не из функции обратного вызова. Также не следует производить остановку при запрещенных прерываниях, или при захваченных ресурсах (объектах синхронизации), так как процедура остановки может занять до 100 ms. Если аварийная или другая ситуация, требующая выполнения остановки DDK, обнаруживается в функции обратного вызова, при занятых (захваченных) ресурсах, или при обработке прерывания, следует запретить генерацию аппаратных прерываний вызовом функции TAU32_DisableInterrupts(). Затем закончить текущую обработку, освободить занятые (захваченные) ресурсы, и после этого произвести остановку. Например, в случае ОС Windows, остановка должна производиться при IRQL = PASSIVE_LEVEL (см. Windows DDK).

Остановка DDK и аппаратной части адаптера выполняется функцией TAU32_DestructiveHalt(). При необходимости функция остановки может обратным вызовом уведомлять ваш код об отмене каждого из запросов, которые были «в ведении» DDK на момент остановки. Такая возможность поддержки единообразного цикла жизни запросов, может быть удобной при необходимости освобождения распределенной памяти и других ресурсов, занимаемых при формировании запросов.

Обслуживание прерываний

Как уже отмечалось, большинство действий DDK выполняет во время обработки аппаратных прерываний генерируемых адаптером, асинхронно по отношению к коду пользователя обращающемуся к DDK. Всю необходимую для этого логику и управляющие структуры DDK скрывает от вас, и организует самостоятельно.

Обслуживание (обработка) аппаратных прерываний производится функцией TAU32_HandleInterrupt(). Для возможности разделения линии IRQ с другими PCI-устройствами, в аппаратной части адаптера предусмотрен соответствующий регистр статуса. Это позволяет обработчику прерываний реагировать только на «свои» прерывания и пропускать «чужие». Соответственно для возможности полноценного взаимодействия с операционной системой функция TAU32_HandleInterrupt() возвращает ненулевой результат, если при вызове имело место аппаратное прерывание от адаптера.

При обработке аппаратных прерываний DDK может многократно вызывать различные обратные вызовы, запускать и отменять принятые к исполнению запросы. Соответственно все предоставленные вами функции обратных вызовов должны быть готовы к тому, что их в любой момент времени может вызвать обработчик прерываний, так же как любая вызванная вами функция DDK.

Обработчик прерываний никогда не должен прерывать вызванные вами функции DDK. Эту синхронизацию должен обеспечивать ваш код, во взаимодействии со средствами операционной системы. Например, в ОС Windows, все вызовы DDK должны производится через системную функцию KeSynchronyzeExecution(). Отступление от этого правило допустимо только когда генерация аппаратных прерываний адаптером запрещена или невозможна. Поэтому существует несколько специальных случаев:

- Функция TAU32_Initialize() не нуждается в синхронизации, поскольку до её вызова генерация прерываний невозможна;
- Функция TAU32_EnableInterrupts(), но только в том случае, если до её вызова генерация прерываний была запрещена;

В случае если в целевой операционной системе обработка аппаратных прерываний невозможна (не реализована), DDK может работать в режиме поллинга (периодического опроса). Для этого достаточно вызывать функцию TAU32_HandleInterrupt(). При этом рекомендуемый период опроса составляет от 125 µs до 2 ms. Увеличение периода опроса свыше 2 ms может приводить к переполнению внутренних очередей событий и к сложным для восстановления ошибкам. Сокращение периода опроса менее 125 µs не окажет какого-либо значимого влияния.

Для нормальной работы DDK должен иметь возможность своевременно обрабатывать аппаратные прерывания. При этом время задержки от момента генерации прерывания до его обслуживания, по возможности не должно превышать 2 ms. В случае крайней необходимости генерация аппаратных прерываний может быть приостановлена функции TAU32_DisableInterrupts(), и затем восстановлена вызо-

вом TAU32_EnableInterrupts(). В частности, именно так следует поступать при необходимости переключения на другую линию IRQ. Для минимизации негативных последствий связанных с невозможностью обработки прерываний, следует приостанавливать приём/передачу по всем активным каналам (остановить их).

В случае если задействован низкоуровневый побайтовый обмен Sa-битами с точностью до одного фрейма E1, задержка более чем на 125 μs может привести к потере принимаемых значений Sa-бит, и «заморозке» (повторной передачи) передаваемых значений.

При длительной задержке в обработке прерываний возможны следующие аварийные ситуации:

- Пауза в обновлении (индикации) актуального статуса интерфейсов E1;
- Пауза в процессе изменения режимов аппаратной части, если запущен запрос на изменение режима (конфигурации);
- «Заморозка» передаваемой сигнализации CAS, при включенном низкоуровневом обмене;
- «Заморозка» значений передаваемых Sa-битов, при включенном низкоуровневом обмене;
- Пауза в передаче информации по «служебным» каналам образованными Sa-битами (возможно с нарушением пакетной структуры HDLC), если «служебные» каналы включены;
- Потеря принимаемой сигнализации CAS, при включенном низкоуровневом обмене;
- Потеря значений принимаемых Sa-битов, при включенном низкоуровневом обмене;
- Потеря данных принимаемых по «служебным» каналам образованными Sa-битами, если «служебные» каналы включены;
- Пауза в формировании и обновлении сигнализации CAS (до 4 мультифреймов после восстановления), при включенном высокоуровневом обмене;
- Пауза в формировании и обновлении передаваемых значениях Sa-бит для всех интерфейсов E1, при включенном высокоуровневом обмене;
- Пауза в передаче данных по логическим каналам;
- Потеря данных принимаемых по логическим каналам;
- Переполнение внутренних очередей событий приёма/передачи;

При переполнении внутренних очередей событий приёма/передачи, для гарантировано полного восстановления, необходима отмена (и перезапуск) всех активных запросов чтения/записи.

Темп генерации аппаратных прерываний

В зависимости от выбранных режимов работы адаптера темп генерации им аппаратных прерываний может сильно варьироваться. Для оценки можно пользоваться следующей схемой:

- Каждый включенный интерфейс E1 генерирует по 32 прерывания в секунду;
- Каждый включенный интерфейс E1 генерирует прерывание при изменении статуса линии;
- При завершении приёма в, или передачи из предоставленного пользовательского буфера генерируется прерывание;
- Если на интерфейсе E1 задействован низкоуровневый побайтовый обмен CAS или FAS на базе сверхциклов E1, то дополнительно будет генерироваться 500 прерываний в секунду (каждые 2 ms);
- Если на интерфейсе E1 задействован низкоуровневый побайтовый обмен FAS на базе кадров (циклов) E1, будет генерироваться 4000 прерываний в секунду (каждые 250 μs);

Кроме этого, прерывания будут генерироваться в процессе актуализации изменений в конфигурации логических каналов, до 16 прерываний на каждый из затронутых каналов. Дополнительно прерывания могут генерироваться при нарушении «непрерывности» потока запросов ввода-вывода и их отмене.

Обработчик прерываний, реализованный в DDK, обрабатывает все прерывания сгенерированные аппаратурой адаптера, как до вызова обработчика, так и во время его выполнения. Поэтому, в случае «наложения» во времени нескольких аппаратных прерываний, обработчик обработает их все, и реальный темп поступления прерываний (вызовов обработчика) может быть несколько меньше.

Приём и передача данных

Как уже отмечалось, DDK не производит копирование принятых или передаваемых данных во внутренние буфера. Весь обмен с приёмопередатчиком через PCI-шину организуется непосредственно через буфера, предоставляемые ваши кодом. Такой подход позволяет минимизировать нагрузку на шину и центральный процессор, но с другой стороны накладывает определенные ограничения:

- Ваш код должен своевременно обеспечивать DDK буферами, указывая при этом их физический, а не виртуальный адрес;
- Предоставляемые буфера должны быть физически непрерывны (последовательны) в оперативной памяти и начинаться с границы машинного слова (32 бита, т.е. адрес начала буфера должен быть кратен 4);
- Предоставляемые буфера должны иметь длину кратную размеру машинного слова, не могут быть меньше машинного слова (32 бита или 4 байта);
- Предоставляемые буфера должны быть доступны с 32-разрядной шины PCI;

Обеспечение DDK буферами, для обмена данными, организуется через очередь запросов ввода-вывода. В запросах на приём/передачу данных задаются физические адреса оперативной памяти и соответственно размер приёмного буфера, или размер передаваемой порции данных. Из этих запросов DDK стремится организовать цепочки буферов для непрерывного обмена данными. Отсюда следует, что приём (и передача) информации будет производиться, только если соответствующие очереди запросов не пусты.

Запросы на передачу завершаются при обнаружении ошибки передачи, либо по завершению загрузки данных в аппаратные FIFO буфера адаптера. При запуске канала на передачу укладка данных гарантированно начнется с ближайшего, младшего по номеру (первого в группе) канального интервала ассоциированного с данным каналом. Другими словами, в «прозрачном» режиме, первый байт первого буфера для передачи будет **гарантированно** размещен в младшем по номеру канальном интервале приёмо-передатчика, начиная с текущего или следующего кадра E1, и таким образом поток передаваемых данных будет синхронизирован с границей кадров E1.

Запросы на приём информации завершаются при обнаружении ошибки приёма, либо после записи в оперативную память последнего байта принятого пакета (для HDLC и других протоколов подразумевающих пакетный режим передачи), или по заполнению предоставленного буфера. При запуске канала на приём укладка данных гарантированно начнется с ближайшего, младшего по номеру (первого в группе) канального интервала ассоциированного с данным каналом. Другими словами, в «прозрачном» режиме, первый байт первого буфера для приёма данных начнёт **гарантированно** формироваться из младшего по номеру канального интервала приёмо-передатчика, начиная с текущего или следующего кадра E1, и таким образом поток принимаемых данных будет синхронизирован с границей кадров E1.

Внимание! Такое, описанное выше, «выравнивание» данных гарантируется только в DDK начиная с версии 1.2, благодаря реализации обхода ошибки дизайна контроллера Infineon MUNICH32X.

Для того чтобы DDK мог гарантированно организовать очереди буферов приёма/передачи достаточные для непрерывного обмена информацией, в каждой из очередей запросов приёма/передачи должно находиться не менее 4 запросов, и с каждым из запросов связан буфер размером не менее 16 байт.

Блок обслуживания сигнализации CAS

При включении блока обслуживания сигнализации по выделенным каналам (CAS), он «врезается» в основной поток данных между приёмопередатчиком и кросс-коннектором. Блок обслуживания CAS «переключает» 16ый канальный интервал приёмопередатчика от кросс-коннектора к себе, и одновременно подключается к шине сигнализации кросс-коннектора.

Блок обслуживания CAS принимает от кросс-коннектора биты сигнализации параллельно основному потоку данных, преобразует их в 16^{ти} байтные пакеты

стандартной структуры (см. ниже), и подставляет в основной поток данных к приёмопередатчику, на место 16го канального интервала.

Соответственно со стороны приёмопередатчика блок обслуживания CAS также принимает данные, выделяя их из 16го канального интервала. Затем распознает 16ти байтные пакеты определенной структуры (см. ниже) и преобразует в биты сигнализации, передаваемые по отдельной шине к кросс-коннектору.

Пакеты данных обрабатываемые блоком обслуживания CAS состоят из 16 байт и имеют следующую структуру (подобно описанной в рекомендации G.704):

	MSB 7	6	5	4	3	2	1	LSB 0
Номер байта	**Информация / Значение**							
0	константа 0x0B							
1	ABCD для №1				ABCD для №16			
2	ABCD для №2				ABCD для №17			
3	ABCD для №3				ABCD для №18			
4	ABCD для №4				ABCD для №19			
5	ABCD для №5				ABCD для №20			
6	ABCD для №6				ABCD для №21			
7	ABCD для №7				ABCD для №22			
8	ABCD для №8				ABCD для №23			
9	ABCD для №9				ABCD для №24			
10	ABCD для №10				ABCD для №25			
11	ABCD для №11				ABCD для №26			
12	ABCD для №12				ABCD для №27			
13	ABCD для №13				ABCD для №28			
14	ABCD для №14				ABCD для №29			
15	ABCD для №15				ABCD для №30			

Другими словами, каждый байт, кроме нулевого, содержит информацию о двух ниблах сигнализации. В старших ниблах (старшие половины байт) расположены биты сигнализации для каналов 1...15, а в младших ниблах (младшие половины байт) для каналов 16...30. При этом бит «A» всегда старше бита «D».

В нулевом байте всегда расположена константа 0x0B, это значение используется для поиска начала блоков в непрерывном потоке байт, передаваемых в 16ом канальном интервале. Несложно подсчитать, что при таком формате темп поступления сигнализации и темп обмена с приёмопередатчиком совпадают.

Блок обслуживания CAS, выделяет пакеты данных из 16го канального интервала, синхронизируясь по константе 0x0B. Синхронизация считается установленной, если в получаемом потоке байт константа встретилась с интервалом в 16 байт. Синхронизация считается потерянной, если вместо ожидаемого ключевого байта, в его предполагаемой позиции, принято другое значение. Ваш код, при приёме сигнали-

зации CAS также должен синхронизироваться и выделять блоки данных подобным методом.

При потере синхронизации блок обслуживания CAS «замораживает» на передачу в кросс-коннектор последние успешно принятые значения. DDK отображает состояние захвата синхронизации при получении блоков от приёмопередатчика в статусе адаптера (см. «TAU32_STATUS»).

Коммутатор Sa-бит

При включении коммутатора Sa-бит, он «врезается» в основной поток данных между приёмопередатчиком и кросс-коннектором. Коммутатор Sa-бит «переключает» 0ой канальный интервал приёмопередатчика от кросс-коннектора к себе, и одновременно подключается к обоим интерфейсам E1.

Коммутатор Sa-бит передает в приёмопередатчик, и получает обратно значения Sa-бит для обоих интерфейсов E1. Но при коммутации позволяет «пробрасывать» Sa-биты из одного интерфейса E1 к другому и наоборот, а также устанавливать их в ноль без использования 0го канального интервала приёмопередатчика.

Коммутатор принимает значения Sa-бит с интерфейсов E1, формирует из них байты (см. ниже), и вставляет в 0ой канальный интервал основного потока данных от кросс-коннектора к приёмопередатчику. Соответственно со стороны приёмопередатчика коммутатор Sa-бит также принимает данные, выделяя их из 0го канального интервала. Затем логически распределяет между (см. ниже) интерфейсами, и передает последовательно с началом очередного нечетного (NAF) кадра E1.

Для распределения потока байтов между интерфейсами, коммутатор Sa-бит использует значение старшего бита. Если старший бит равен нулю, то байт отправляется в первый (E1/0) интерфейс, иначе во второй (E1/1). При формировании потока байт для 0го канального интервала, коммутатор использует старший бит аналогичным образом. В результате, при взаимодействии с коммутатором Sa-бит, используется следующий формат байтов 0го канального интервала приёмопередатчика:

	MSB							LSB
Номер бита:	7	6	5	4	3	2	1	0
	n	*	*	Sa4	Sa5	Sa6	Sa7	Sa8

«n» – определяет интерфейс E1 («0» для E1/0, «1» для E1/1);
«*» – зарезервировано, при приёме ваш код не должен предполагать какие-либо определенные значения, а при передаче устанавливать в «0»;

Низкоуровневый обмен

В дополнение к высокоуровневому обмену данными на основе логических каналов, в DDK реализовано несколько режимов низкоуровневого побайтного обмена на базе FIFO.

Режимы низкоуровневого обмена предназначены для использования в следующих случаях:

- Приём/передача данных сигнализации по выделенным каналам (CAS) непосредственно с интерфейсов E1;
- Приём/передача значений битов Sa4-Sa8, Ra и Si расположенных в 0ом канальном интервале нечетных (NAF) кадров E1;

В режимах низкоуровневого обмена, порядок битов в байте всегда соответствует «Network order», старший бит байта передаётся первым.

Для каждого интерфейса E1 предусмотрены четыре FIFO-буфера, по два на приём и на передачу, для сигнализации CAS и для потока Sa-битов. Каждому из четырех «типов» FIFO-буферов в DDK сопоставлен идентификатор из набора «TAU32_FIFO_ID». Также каждому из четырех FIFO, соответствует счетчик, который считает ситуации, когда FIFO либо не может вместить новую порцию принятых данных, либо не содержит достаточно данных для передачи.

Обмен с FIFO, и далее с интерфейсами E1, происходит всегда на байтовом (не битовом) уровне, при этом отдельные биты могут не используются, а в данных для передачи должна соблюдаться соответствующая структура.

Обмен информации между FIFO и интерфейсами E1 происходит при обработке аппаратных прерываний адаптера. Для обмена информации с FIFO-буферами, вашему коду доступна группа функция TAU32_Fifo*(). Кроме этого, в DDK реализован механизм управляемого уведомления вашего кода об изменении уровня FIFO-буферов, через функции обратных вызовов. Указанная вами функция, будет вызываться DDK каждый раз, когда уровень FIFO пересечет заданную вами границу.

Всего в DDK реализовано три режима низкоуровневого обмена, все режимы включаются при конфигурировании интерфейсов E1 соответствующими флагами.

Режим обмена данными сигнализации CAS блоками по 16 байт (**TAU32_cas_io**). Каждый байт соответствует октету 16го канального интервала. Для передачи, ваш код должен подготавливать 16-байтные блоки данных сигнализации в соответствии с рекомендацией ITU-T G.704. При приёме вам необходимо выбирать данные строго по 16 байт, либо производить поиск начала сверхцикла в соответствии G.732 §5.2

№	MSB							LSB
0	0	0	0	0	X	**Y**	X	X
1	A-1	B-1	C-1	D-1	A-16	B-16	C-16	D-16
2	A-2	B-2	C-2	D-2	A-17	B-17	C-17	D-17
…	…	…	…	…	…	…	…	…
16	A-15	B-15	C-15	D-15	A-30	B-30	C-30	D-30

При включении режима TAU32_cas_io аппаратные прерывания будут генерироваться не реже 500 раз в секунду.

Режим побайтного обмена битами Sa4-Sa8, Ra и опционально Si (**TAU32_fas_io**). Использование бита Si возможно только если на интерфейсе E1 не задействована сверхцикловая синхронизация CRC4. Обмен происходит побайтно, каждый байт соответствует октету $0^{го}$ канального интервала в нечетных (NAF) фреймах E1. При включении режима аппаратные прерывания будут генерироваться не реже 4000 раз в секунду.

Режим обмена битами Sa4-Sa8, Ra и опционально Si блоками по 8 байт (**TAU32_fas8_io**). Использование битов Si возможно только если на интерфейсе E1 не задействована сверхцикловая синхронизация CRC4. Обмен происходит блоками по 8 байт. Каждый байт блока образуется из 8 значений соответствующего бита $0^{го}$ канального интервала из последовательно принятых 16 фреймов E1.

№ байта в блоке	Соответствующие биты $0^{го}$ канального интервала
0	восемь Si битов, из 8 четных (AF) фреймов E1
1	восемь Si битов, из 8 нечетных (NAF) фреймов
2	восемь Ra битов, из 8 нечетных (NAF) фреймов E1
3	восемь Sa4 битов, из 8 нечетных (NAF) фреймов E1
4	восемь Sa5 битов, из 8 нечетных (NAF) фреймов E1
5	восемь Sa6 битов, из 8 нечетных (NAF) фреймов E1
6	восемь Sa7 битов, из 8 нечетных (NAF) фреймов E1
7	восемь Sa8 битов, из 8 нечетных (NAF) фреймов E1

При включении режима TAU32_fas8_io аппаратные прерывания будут генерироваться не реже 500 раз в секунду.

Одновременно из режимов TAU32_fas8_io и TAU32_fas_io может быть включен только один.

При выключении интерфейса E1 (переводе в состояние TAU32_LineOff) очищаются его «передающие» FIFO-буфера. При включении интерфейса E1 (выводе из состояния TAU32_LineOff) очищаются его «приёмные» FIFO-буфера.

Неявные действия обслуживания E1

При обнаружении аварийных ситуаций каналов E1, а также соответствующих настройках других узлов адаптера, DDK будет выполнять определенные действия, связанные с замещением передаваемых и в некоторых случаях принимаемых данных отдельных канальных интервалов E1.

Для неструктурированного режима E1 и режимов с занижением скорости:

- Если хотя бы по одному из канальных интервалов, образующих неструктурированный поток данных, не производится передача со стороны приёмопередатчика, то в линию E1 будет передаваться сигнал AIS (все единицы);

Для структурированных режимов E1 (с цикловой структурой согласно G.704):

- В канальных интервалах, источники данных для которых имеют аварийное состояние, передаются все единицы (код 0xFF);
- В канальных интервалах, не имеющих источника данных, передаётся «код бездействия» («Idle Code»). По умолчанию его значение устанавливается равным 0xD5, и может быть изменено средствами DDK независимо для каждого канального интервала;
- При обнаружении на интерфейсе E1 аварийных ситуаций «RCL (Receive Carrier Lost)», «RFAS (FAS Frame Sync Loss)», или если при включенной сверхцикловой синхронизации CRC4 она не может быть найдена в течение 128 ms, то в исходящем потоке будет установлен сигнал «Remote Alarm» (если это не запрещено в конфигурации интерфейса E1);
- Если не задействована сверхцикловая синхронизация CRC4 и режимы низкоуровневого обмена, то в исходящем потоке передаются биты Si равные «1»;
- Если не задействован коммутатор Sa-бит и режимы низкоуровневого обмена, то в исходящем потоке передаются биты Sa4, Sa5, Sa6, Sa7, Sa8 равные «1»;
- Если все канальные интервалы образующие исходящий поток E1 имеют аварийные источники, то опционально в линию может выдаваться сигнал AIS (все единицы);

Для структурированных режимов со сверхцикловой структурой сигнализации по выделенным каналам (CAS):

- В битах сигнализации CAS, соответствующих канальным интервалам, интервалах, источники данных для которых имеют аварийное состояние, передаются все единицы (код 0xF);
- В битах сигнализации CAS, соответствующих канальным интервалам не имеющих источника данных, передаётся ABCD-комбинация «1101» (код 0xD);
- При обнаружении на интерфейсе E1 аварийных ситуаций «RSA1 (Receive Signaling All Ones)» или «RCAS (CAS Multiframe Sync Loss)» в исходящем потоке сигнализации CAS будет установлен сигнал «Distant Multiframe Alarm» (если это не запрещено в конфигурации интерфейса E1);

При использовании режимов высокоуровневого обмена:

- Если задействован режим высокоуровневого обмена данными сигнализации CAS через 16ый канальный интервал приёмопередатчика и передача в этом канальном интервале не ведется, то сигнализация со стороны приёмопередатчика считается аварийной, и при её передаче в интерфейсы E1 будет производиться соответствующая подстановка;

Если задействован коммутатор Sa-бит и данные из 0го канального интервала приёмопередатчика скоммутированы в интерфейс E1, но при этом передача в 0ом канальном интервале не ведется, то в исходящем потоке соответствующих интерфейсов будут передаваться Sa-биты равные «1»;

Аварийными источниками считаются канальные интервалы интерфейсов E1 с аварийными ситуациями: RCL, RUA1, RFAS, RCRC4 (см. «

Критерии аварийных ситуаций E1»). А также канальные интервалы приёмопередатчика, по которым не производится передача данных. Канальные интервалы, задействованные в неструктурированном режиме, а также 0ой и 16ый канальные интервалы, соответственно при включенных коммутаторе Sa-бит и блоке обслуживания сигнализации CAS, считаются аварийными по отношению к другим узлам адаптера.

Аварийными источниками сигнализации CAS считаются интерфейсы E1 с аварийными ситуациями: RCL, RUA1, RFAS, RCRC4, RCAS (см. «

Критерии аварийных ситуаций E1»). А также блок обслуживания сигнализации CAS, если со стороны приёмопередатчика не ведется передача в 16ом канальном интервале.

Типы данных и переносимость

В исходных текстах DDK и заголовочных файлах используются следующие типы данных (стандарт языка C99):

unsigned __int8 – беззнаковое целое длиной в 8 бит (один байт). В устаревших компиляторах C/C++ этому типу соответствует «unsigned char».

unsigned __int16 – беззнаковое целое длиной в 16 бит (два байта). В устаревших компиляторах C/C++ C этому типу, в большинстве случаев, соответствует «unsigned short».

unsigned __int32 – беззнаковое целое длиной в 32 бита (четыре байта). В устаревших компиляторах C/C++ этому типу, в большинстве случаев, соответствует «unsigned long».

unsigned __int64 – беззнаковое целое длиной в 64 бита (восемь байт). В некоторых устаревших компиляторах C/C++ этому типу соответствует «unsigned long long».

В тех случаях, когда нет жестких требований к размеру (длине) объектов, используется типы unsigned (unsigned int) и int (signed int). При этом предполагается, что компилятор сам подберёт наиболее удобное для него (и аппаратной платформы) представление данных.

Возможно, что для 16-битных платформ в ряде случаев придется явно заменять типы unsigned и int, соответственно на unsigned long и long, либо на unsigned short и short. Однако функционирование DDK на 16-битных платформах или 8-битных не предполагается, и поэтому такие случаи не рассматривались и не анализировались.

Использование констант, структур и функций

При использовании DDK не следует:

- использовать непосредственно числовые представления каких-либо определенных констант;
- опираться на порядок следования элементов структур;
- опираться на размер определенных типов данных и структур;
- использовать какие-либо недокументированные возможности;
- использовать какие-либо эффекты вызова функций DDK, кроме явно определенных;

Все перечисленные элементы могут меняться при обновлении текущей версии DDK или выходе новых версий. Нарушая эти рекомендации, вы потеряете совместимость на уровне исходных текстов.

Проверка параметров и сбои

Функции DDK производят только базовую проверку получаемых от вас параметров и только тех, значения которых имеют заранее известные пределы допустимых значений.

Если вы, по ошибке или специально, передадите в функции DDK заведомо неверные параметры (например, адреса ввода-вывода), то, скорее всего, это приведет к фатальному сбою и перезагрузке компьютера.

Справочник

Структуры и константы

Идентификаторы адаптера на шине PCI

```
#define TAU32_PCI_VENDOR_ID      0x…
#define TAU32_PCI_DEVICE_ID      0x…
#define TAU32_PCI_IO_BAR1_SIZE   0x…
#define TAU32_PCI_RESET_ADDRESS  0x…
#define TAU32_PCI_RESET_ON       0x…
#define TAU32_PCI_RESET_OFF      0x…
```

В DDK определены три соответствующие константы:

- ***TAU32_PCI_VENDOR_ID*** – значение «PCI Vendor Identifier» для адаптера Tau32;
- ***TAU32_PCI_DEVICE_ID*** – значение «PCI Device Identifier» для адаптера Tau32;
- ***TAU32_PCI_IO_BAR1_SIZE*** – размер в байтах отображаемого пространства ввода-вывода для адаптера Tau32;
- ***TAU32_PCI_RESET_ADDRESS*** – адрес PCI-регистра адаптера для аппаратного сброса приёмопередатчика (контроллера HDLC);
- ***TAU32_PCI_RESET_ON*** – значение, которое необходимо записать в PCI-регистр сбора для подачи сигнала полного аппаратного сброса приёмопередатчика;
- ***TAU32_PCI_RESET_OFF*** – значение, которое необходимо записать в PCI-регистр сбора для снятия сигнала аппаратного сброса приёмопередатчика;

Внимание! Вследствие аппаратной ошибки в контроллере Infineon MUNICH32X при сбросе приёмопередатчика возможна генерация неожидаемого аппаратного прерывания. Для обхода данной ситуация в DDK начиная с версии 1.2 введена функция «TAU32_BeforeReset()».

TAU32_Controller{}, TAU32_ControllerObjectSize

```
typedef struct tag_TAU32_Controller TAU32_Controller;
extern unsigned const TAU32_ControllerObjectSize;
```

TAU32_Controller – это основная внутренняя структура DDK. С каждым обслуживаемым адаптером Tau32, при его инициализации связывается экземпляр TAU32_Controller и TAU32_UserContext. DDK полностью скрывает от вас внутреннее устройство этой структуры, всё что о ней известно – её размер, хранящийся в константе TAU32_ControllerObjectSize.

При вызове любой функции DDK, кроме TAU32_Initialize(), первым параметром передается указатель на соответствующий экземпляр TAU32_Controller.

При инициализации DDK вам необходимо выделить память под эту структуру. Необходимо чтобы этот участок памяти был резидентным, выровненным на границу машинного слова (32 бита), и физически непрерывным, несмотря на страничную и/или виртуальную организацию оперативной памяти в целевой операционной системе.

TAU32_UserContext{}

```
typedef struct tag_TAU32_UserContext
{
    TAU32_Controller *pControllerObject;
    PCI_PHYSICAL_ADDRESS ControllerObjectPhysicalAddress;
    void *PciBar1VirtualAddress;
    TAU32_NotifyCallback pErrorNotifyCallback;
    TAU32_NotifyCallback pStatusNotifyCallback;

    int Model;
    int Interfaces;
    unsigned InitErrors;
    unsigned __int32 DeadBits;
    unsigned AdapterStatus;
    unsigned CasIoLofCount;
    unsigned E1IntLostCount;
    TAU32_E1_State InterfacesInfo[2];

    // вы можете добавить необходимые вам элементы здесь,
    // определив TAU32_UserContext_Add
} TAU32_UserContext;
```

Структура содержит элементы необходимые для организации взаимодействия DDK с вашим кодом. С каждым обслуживаемым адаптером Tau32, при его инициализации связывается экземпляр TAU32_UserContext, и через него экземпляр TAU32_Controller.

При инициализации DDK вы должны выделить память под эту структуру, и заполнить следующие поля:

- ***pControllerObject*** – виртуальный адрес экземпляра TAU32_Controller;
- ***ControllerObjectPhysicalAddress*** – физический адрес экземпляра TAU32_Controller;
- ***PciBar1VirtualAddress*** – виртуальный адрес регистров ввода/вывода адаптера, отображенных в оперативную память
- ***pErrorNotifyCallback*** – «NULL» либо адрес функции обратного вызова, которой будет сообщаться об ошибках приёма/передачи и других аварийных ситуациях приёмопередатчика;
- ***pStatusNotifyCallback*** – «NULL» либо адрес функции обратного вызова, которой будет сообщаться об изменении статуса адаптера и интерфейсов E1;

Участок памяти, выделяемый для TAU32_Controller, должен быть резидентным, выровненным на границу машинного слова (32 бита), и физически последовательным (непрерывным), т.е. располагаться по физически строго последовательным адресам оперативной памяти (см. MmAllocateContiguousMemory() в Windows DDK).

Отображение регистров ввода/вывода должно быть произведено с отключением кэширования, а также обратной, пакетной и комбинированной записи со стороны центрального процессора, или всех процессоров, которые будут вызывать функции DDK (см. MmMapIoSpace() в Windows DDK).

Указатель на заполненный экземпляр TAU32_UserContext передается первым параметром функции TAU32_Initialize(), которая производит инициализацию DDK, инициализацию адаптера и его базовое тестирование.

В качестве результата выполнения, функция TAU32_Initialize(), возвращает булево значение, и устанавливает следующие поля структуры TAU32_UserContext:

- ***Model*** – модель адаптера Tau32, может принимать одно из значений TAU32_MODELS;
- ***Interfaces*** – количество интерфейсов E1, доступных на данной модели адаптера;
- ***InitErrors*** – логическое объединение битовых флагов, индицирующих проблемы обнаруженные при инициализации;
- ***DeadBits*** – дополнительная информация об обнаруженных аппаратных не-исправностях, предназначено для использования только персоналом «Кроникс»;

Элементы Status, CasIoLofCount, E1IntLostCount и InterfacesInfo будут обновляться DDK в процессе работы. Вы можете обращаться к этим элементам непосредственно. В процессе работы эти поля будут содержать следующую информацию:

- ***AdapterStatus*** – статус адаптера, комбинацию битовых флагов TAU32_STATUS;
- ***CasIoLofCount*** – счетчик ошибок формирователя сигнализации CAS;
- ***E1IntLostCount*** – счетчик пропущенных прерываний при задействовании режимов низкоуровневого побайтного обмена;
- ***InterfacesInfo*** – массив структур TAU32_E1_State{}, каждый элемент соответствует интерфейсу E1;

Элементы pErrorNotifyCallback и pStatusNotifyCallback можно обновлять по необходимости в любое время.

TAU32_UserContext_Add

Для удобства, в DDK предусмотрена возможность дополнять структуру TAU32_UserContext элементами необходимыми вашему коду. Для этого вам необходимо определить макрос TAU32_UserContext_Add, например так:

```
#define TAU32_UserContext_Add \
    int Id; \
    void *pTag, *pTag2, *pTag3;
```

За дополнительной информацией обращайтесь к разделам «Инициализация и запуск» и «TAU32_Initialize()».

TAU32_INIT_ERRORS

```
#define TAU32_IE_OK             0x…
#define TAU32_IE_FIRMWARE       0x…
#define TAU32_IE_MODEL          0x…
#define TAU32_IE_E1_A           0x…
#define TAU32_IE_E1_B           0x…
#define TAU32_IE_INTERNAL_BUS   0x…
#define TAU32_IE_HDLC           0x…
#define TAU32_IE_ADPCM          0x…
#define TAU32_IE_CLOCK          0x…
#define TAU32_IE_DXC            0x…
#define TAU32_IE_XIRQ           0x…
```

Набор констант определяющих битовые флаги, которые соответствуют возможным проблемам, обнаруживаемым при инициализации и базовом тестировании адаптера (поле InitErrors структуры TAU32_UserContext):

- ***TAU32_IE_OK*** – нет ошибок;
- ***TAU32_IE_FIRMWARE*** – ошибка загрузки firmware;
- ***TAU32_IE_MODEL*** – ошибка определения модели адаптера, либо обнаружена модель, которая не поддерживается DDK;
- ***TAU32_IE_E1_A*** – неисправность первого (E1/0) интерфейса E1;
- ***TAU32_IE_E1_B*** – неисправность второго (E1/1) интерфейса E1;
- ***TAU32_IE_INTERNAL_BUS*** – неисправность внутренней шины;
- ***TAU32_IE_HDLC*** – неисправность контроллера HDLC;
- ***TAU32_IE_ADPCM*** – неисправность компрессора ADPCM;
- ***TAU32_IE_CLOCK*** – неисправность тактового генератора;
- ***TAU32_IE_DXC*** – неисправность одного из кросс-коннекторов;
- ***TAU32_IE_XIRQ*** – неисправность линии IRQ от контроллеров E1;

За дополнительной информацией обращайтесь к разделам «Инициализация и запуск» и «TAU32_Initialize()».

TAU32_MODELS

```
#define TAU32_ERROR      0x…
#define TAU32_UNKNOWN    0x…
#define TAU32_BASE       0x…
#define TAU32_LITE       0x…
#define TAU32_ADPCM      0x…
```

Набор констант определяющих модели адаптера Tau32 (возможные значения поля Model структуры TAU32_UserContext):

- ***TAU32_ERROR*** – ошибка при определении модели адаптера;
- ***TAU32_UNKNOWN*** – неизвестная для используемой версии DDK модель адаптера;
- ***TAU32_BASE*** – базовая модель, два интерфейса E1;
- ***TAU32_LITE*** – облегченная модель, один интерфейс E1;

За дополнительной информацией обращайтесь к разделам «Инициализация и запуск» и «TAU32_Initialize()».

TAU32_tsc{}

```
struct tag_TAU32_tsc
{
    unsigned __int32 osc, sync;
} TAU32_tsc;
```

TAU32_tsc определяет структуру для получения значений счетчиков таковых импульсов опорного тактового генератора и единого внутреннего источника синхронизации адаптера. Данная структура заполняется функцией TAU32_ReadTsc(), при этом поля имеют следующие значения:

- ***osc*** – 32-битное значение счетчика тактовых импульсов внутреннего генератора адаптера;
- ***sync*** – 32-битное значение счетчика тактовых импульсов на внутренней шине синхронизации (от текущего источника синхронизации);

Опорный тактовый генератор вырабатывает частоту 2048 KHz ±50 ppm. Каждый такт единой внутренней синхронизации соответствует одному принятому/переданному биту в потоках E1.

Оба счетчика считают тактовые импульсы по модулю 2^{32} с момента инициализации адаптера, но при сбросе и/или запуске адаптера не устанавливаются в «0».

Счетчик «sync» считает импульсы на внутренней шине синхронизации. При смене первоисточника синхронизации функцией TAU32_SetSyncMode() счетчики не сбрасывается.

Для дополнительной информации см. описание функций «TAU32_ReadTsc()» и «TAU32_SetSyncMode()», а также раздел «Устройство адаптера».

TAU32_UserRequest{}

```
typedef struct tag_TAU32_UserRequest
{
    void *pInternal;
    unsigned Command;
#if defined(_NTDDK_)
    KDPC CallbackDpc;
    void SetupCallback(
        PKDEFERRED_ROUTINE DeferredCallbackRoutine,
        void* pContext);
    void SetupCallback(TAU32_RequestCallback pCallback);
#else
    TAU32_RequestCallback pCallback;
#endif
    unsigned __int32 ErrorCode;
    union
    {
        unsigned ChannelNumber;
        struct
        {
            unsigned Channel;
            unsigned __int32 Config;
            unsigned __int32 AssignedTsMask;
        } ChannelConfig;
        struct
        {
            int Interface;
            unsigned __int32 Config;
            unsigned __int32 UnframedTsMask;
        } InterfaceConfig;
        struct
        {
            unsigned Channel;
            PCI_PHYSICAL_ADDRESS PhysicalDataAddress;
            unsigned DataLength;
            unsigned Transmitted;
        } Tx;
        struct
        {
            unsigned Channel;
            PCI_PHYSICAL_ADDRESS PhysicalDataAddress;
            unsigned BufferLength;
            unsigned Received;
```

```
            BOOLEAN FrameEnd;
        } Rx;

        BOOLEAN DigitalLoop;

        union
        {
            TAU32_TimeslotAssignment Complete[TAU32_TIMESLOTS];
            unsigned __int32 Map[TAU32_CHANNELS];
        } TimeslotsAssignment;
    } Io;

    // вы можете добавить необходимые вам элементы здесь,
    // определив TAU32_UserRequest_Add
} TAU32_UserRequest;
```

TAU32_UserRequest определяет структуру всех асинхронных запросов обрабатываемых DDK. Участки оперативной памяти для запросов должны выделяться из резидентной зоны.

Большинство элементов структуры задает те или иные передаваемые параметры. Действия, которые должен выполнить DDK, определяются значением поля Command. Соответственно этим же определяется, какие именно параметры следует передавать, какие поля структуры, и как будут задействованы.

Несмотря на то, что поле Command позволяет комбинировать все команды, лишь некоторые из них могут объединяться в одном запросе (см. «TAU32_REQUEST_COMMANDS»). При объединении нескольких команд в одном запросе, необходимо заполнять все соответствующие параметры. Объединенные команды при этом могут выполняться параллельно и/или последовательно, в зависимости от качественного состава созданного «объединения» (см. ниже).

После отправки запроса к DDK, какие-либо его элементы не должны изменяться вашим кодом. Это может привести к катастрофическим ошибкам в работе DDK.

Общие поля

- ***pInternal*** – указатель, используемый внутри DDK, перед отправкой запроса должен быть установлен в «NULL». Пока запрос выполняется (находится в ведении DDK) этот элемент не равен нулю, и ни в коем случае не должен изменяться вашим кодом. Исключение составляет только ситуация после вызова TAU32_DestructiveHalt(), когда DDK останавливается и внутренние структуры разрушаются;
- ***Command*** – задает действия, которые должен выполнить DDK (см. «TAU32_REQUEST_COMMANDS»). В процессе обработки и выполнения запроса DDK, обновляет это поле, обнуляя отдельные биты с началом выполнения соответствующих команд. Если запрос, скомбинированный из нескольких команд, был отменен или завершился с ошибкой, то ненулевые биты этого поля будут соответствовать командам, к выполнению которых DDK не приступал. Если на момент отмены запрос находился во входящей очереди, то поле Command останется нетронутым;

- ***pCallback*** – задает адрес функции обратного вызова, которая получит уведомление от DDK о завершении выполнения или отмены запроса (см. «TAU32_RequestCallback()»). Этот параметр не может быт равен «NULL»;
- ***ErrorCode*** – логическое объединение битовых флагов «TAU32_ERRORS», индицирующих ошибки, произошедшие при выполнении запроса. При приёме запроса к исполнению, DDK устанавливает это поле в «0», и затем обновляет в процессе выполнения и/или отмены запроса;

Параметры и результаты

Как уже отмечалось, актуальный набор необходимых параметров зависит от команды посылаемой к DDK. Во всех случаях, когда подразумевается использование какого-то конкретного логического канала, необходимо задавать его в поле Io.ChannelNumber. При конфигурировании интерфейсов E1, необходимо задавать поле Io.InterfaceConfig.Interface.

Для команд приёма/передачи в параметрах запроса необходимо указывать физический адрес буфера в оперативной памяти, с которым будет производиться обмен информацией. Как и в случае со структурой TAU32_Controller, этот буфер должен быть физически непрерывным (последовательным) и выровнен на границу машинного слова (32 бита). Длина передаваемой или принимаемой порции данных должна быть не меньше 4 байт.

Далее описаны параметры запросов, необходимые для каждой группы команд.

Приём данных

Выполняется командой TAU32_Rx_Data (см. TAU32_REQUEST_COMMANDS). Длина принимаемой порции данных должна быть не меньше 4 байт. Если логический канал использует канальный протокол предполагающий пакетный режим (например HDLC), то размер принимаемой порции данных (кадра, пакета), не может превышать 8188 байт включая служебную информацию, а размер приёмного буфера должен быть на 4 байта больше максимального размера пакета. Запросы на приём данных не могут быть скомбинированы с запросами на передачу.

Кроме указания номера логического канала и установки поля Command, необходимо заполнить подструктуру Io.Rx:

- ***Io.Rx.PhysicalDataAddress*** – физический адрес буфера для записи принимаемых данных. Буфер должен быть физически непрерывным (последовательным) и выровнен на границу машинного слова (32 бита);
- ***Io.Rx.BufferLength*** – размер буфера и соответственно максимальный размер принимаемой порции данных;

По завершению выполнения, или отмены запроса DDK установит поле ErrorCode в соответствующее значение, а также обновит поля подструктуры Io.Rx:

- ***Io.Rx.Received*** – количество принятых и записанных в буфер байт данных;

- ***Io.Rx.FrameEnd*** – ненулевое значение будет означать, что завершен приём пакета (фрейма) данных;

Передача данных

Выполняется командой TAU32_Tx_Data (см. TAU32_REQUEST_COMMANDS). Длина передаваемой порции данных должна быть не меньше 4 байт. Запросы на передачу данных не могут быть скомбинированы с запросами на приём. Кроме указания номера логического канала и установки поля Command, необходимо заполнить подструктуру Io.Tx:

- ***Io.Tx.PhysicalDataAddress*** – физический адрес буфера с данными для передачи. Буфер должен быть физически непрерывным (последовательным) и выровнен на границу машинного слова (32 бита);
- ***Io.Tx.DataLength*** – размер порции данных для передачи;

Если для логического канала установлен режим предполагающий пакетный режим, то в поле Command может быть установлен бит TAU32_Tx_FrameEnd. В этом случае при завершении передачи, передатчик оформит завершение пакета (кадра, фрейма) в соответствии с установленным канальным протоколом.

По завершению выполнения, или отмены запроса DDK установит поле ***ErrorCode*** в соответствующее значение, а также запишет в поле ***Io.Tx.Transmitted*** количество переданных байтов.

Управление логическими каналами

Под управлением в данном случае понимаются следующие команды:

- Остановка и запуск передачи по логическому каналу;
- Остановка и запуск приёма по логическому каналу;
- Выбор и настройка канального протокола;
- Привязка канальных интервалов E1 к логическому каналу;

Все перечисленные команды могут быть скомбинированы в одном запросе, а также объединены с командами на передачу, либо на приём данных.

При изменении конфигурации логических каналов, остановке и запуске, как приёма, так и передачи, DDK лишь подготавливает свои внутренние структуры. Реальное изменение конфигурации и управление оборудованием производиться только при подаче запроса с установленным битом команды TAU32_Configure_Commit (см. «TAU32_REQUEST_COMMANDS»). Это позволяет задавать и изменять конфигурацию несколькими запросами, и только по завершению манипуляций производить актуализацию изменений.

При актуализации изменений DDK может следовать либо «коротким», либо «длинным» путем. «Короткий путь» означает актуализацию изменений только для одного логического канала, а «длинный путь» – для всех. Если изменения, накопившиеся до подачи команды TAU32_Configure_Commit, затрагивают только один

логический канал, то DDK выбирает «короткий путь», иначе «длинный». Изменение привязки канальных интервалов к логическим каналам всегда инициирует «длинный путь», это связанно с особенностями в работе контроллера MUNICH32X.

Изменение конфигурации логических каналов производиться только в отключенном состоянии. Если запрос изменения конфигурации поступает, при включенной передаче и/или приёме, то DDK автоматически отключит канал перед изменением конфигурации, и после восстановит его режим работы. Соответственно если DDK следует по «длинному пути», то будут остановлены и затем перезапущены все активные логические каналы.

Если изменение конфигурации необходимо одновременно с запуском или остановкой отдельных каналов, то DDK стремится изменить конфигурацию до запуска или после остановки.

Остановка и запуск передачи

Остановка и запуск передачи в логическом канале производится командами TAU32_Tx_Start и TAU32_Tx_Stop соответственно (см. «TAU32_REQUEST_COMMANDS»). В параметрах кроме установки поля Command необходимо указать номера логического канала. Установка бита TAU32_Configure_Commit вызовет актуализацию накопленных изменений в ходе выполнения запроса.

Если обе команды скомбинированы в одном запросе, то сначала производится запуск, и затем остановка передачи. Если команды объединены с запросами на приём или передачу, то команды обмена данными будут выполнены после запуска, и/или перед остановкой.

В связи с особенностями работы контроллера MUNICH32X остановка передачи может генерировать несколько ошибок протокола канального уровня на принимающей стороне. Например, в случае HDLC возможен не только обрыв кадра передаваемого в данный момент, но и генерация одного-двух коротких кадров HDLC. Вопрос минимизации или устранения таких явлений изучается.

Остановка и запуск приёма

Остановка и запуск приёма в логическом канале производится TAU32_Rx_Start и TAU32_Rx_Stop соответственно (см. «TAU32_REQUEST_COMMANDS»). В параметрах кроме установки поля Command необходимо указать номера логического канала. Установка бита TAU32_Configure_Commit вызовет актуализацию накопленных изменений в ходе выполнения запроса.

Если обе команды скомбинированы в одном запросе, то сначала производится запуск, и затем остановка приёма. Если команды объединены с запросами на приём или передачу, то команды обмена данными будут выполнены после запуска, и/или перед остановкой.

Выбор и настройка канального протокола

Выбор и настройка канального протокола производится командой TAU32_Configure_Channel. Кроме указания номера логического канала и установки поля Command, необходимо задать конфигурацию в поле Config. Конфигурация задается комбинацией (логическим объединением) канального протокола (см. «TAU32_CHANNEL_MODES») и необходимых флагов (см. «TAU32_CHANNEL_CONFIG_BITS»).

Установка бита TAU32_Configure_Commit вызовет актуализацию накопленных изменений в ходе выполнения запроса. Команда может объединяться с командами остановки и/или запуска приёма и/или передачи, и с командами приёма или передачи данных (см. Управление логическими каналами).

Привязка канальных интервалов

Существуют три команды осуществляющие привязку канальных интервалов к логическому каналу:

- ***TAU32_Timeslots_Channel*** – осуществляет привязку канальных интервалов выбранных битовой маской TimeslotsAssignment.Mask к указанному логическому каналу. Если какие-либо из выбранных канальных интервалов ранее использовались другим логическим каналам, то DDK произведет переключение;
- ***TAU32_Timeslots_Map*** – осуществляет привязку канальных интервалов, основываясь на значениях массива TimeslotsAssignment.Map. Каждый элемент массива соответствует логическому каналу, а значение задаёт маску привязываемых канальных интервалов;
- ***TAU32_Timeslots_Complete*** – задает полную конфигурацию привязки канальных интервалов к логическим каналам. Конфигурация задается массивом TimeslotsAssignment.Complete (см. «TAU32_TimeslotAssignment{}»);

Установка бита TAU32_Configure_Commit вызовет актуализацию накопленных изменений в ходе выполнения запроса. В запросе может быть задана только одна из этих команд, но при этом объединяться с командами остановки и/или запуска приёма и/или передачи, и с командами приёма или передачи данных (см. Управление логическими каналами).

Конфигурирование интерфейсов E1

Конфигурирование интерфейсов E1 производится командой TAU32_Configure_E1. В параметрах кроме установки Command необходимо в поле Io.InterfaceConfig.Interface задать интерфейс, который будет настраиваться, и его конфигурацию в поле Config (см. «TAU32_INTERFACE_CONFIG_BITS»).

Эта команда не может комбинироваться с какими-либо другими, заданные изменения актуализируются DDK всегда непосредственно в процессе выполнения запроса (бит TAU32_Configure_Commit не имеет значения).

Во всех случаях, кроме включения неструктурированного режима, допускается конфигурировать все интерфейсы, указав Io.InterfaceConfig.Interface = TAU32_E1_ALL (см. «TAU32_INTERFACES»). При переключении в неструктурированный режим с занижением скорости, необходимо дополнительно указывать, какие из канальные интервалов приёмопередатчика следует использовать. Для этого в поле TimeslotsAssignment.Mask необходимо задать соответствующую битовую маску. При включении неструктурированного режима без занижения скорости, DDK автоматически задействует все канальные интервалы приёмопередатчика.

Изменение конфигурации интерфейса E1 может потребовать до 40 ms времени асинхронного выполнения (при переключении из TAU32_LineOff). При перенастройке нескольких интерфейсов время обработки не увеличивается.

TAU32_UserRequest_Add

Для удобства, в DDK предусмотрена возможность дополнять структуру TAU32_UserContext необходимыми вашему коду элементами. Для этого вам необходимо определить макрос TAU32_UserRequest_Add, например так:

```
#define TAU32_UserRequest_Add \
    int Id; \
    void *pTag, *pTag2, *pTag3;
```

За дополнительной информацией обращайтесь к разделам «Обработка запросов» и «TAU32_SubmitRequest()».

TAU32_REQUEST_COMMANDS

```
#define TAU32_Tx_Start              0x…
#define TAU32_Tx_Stop               0x…
#define TAU32_Tx_Data               0x…

#define TAU32_Rx_Start              0x…
#define TAU32_Rx_Stop               0x…
#define TAU32_Rx_Data               0x…

#define TAU32_Configure_Channel     0x…
#define TAU32_Timeslots_Complete    0x…
#define TAU32_Timeslots_Map         0x…
#define TAU32_Timeslots_Channel     0x…
#define TAU32_Configure_Commit      0x…

#define TAU32_Tx_FrameEnd           0x…
#define TAU32_Tx_NoCrc              0x…
#define TAU32_Configure_E1          0x…
```

Набор констант, определяющий команды, посылаемые к DDK в асинхронных запросах. Некоторые команды могут комбинироваться в одном запросе.

- ***TAU32_Tx_Start*** – запустить передачу по указанному логическому каналу. Изменения в конфигурации и состоянии логического канала будут актуализированы при подаче команды TAU32_Configure_Commit;

- ***TAU32_Tx_Stop*** – остановить передачу по указанному логическому каналу. Изменения в конфигурации и состоянии логического канала будут актуализированы при подаче команды TAU32_Configure_Commit;
- ***TAU32_Tx_Data*** – передать (поставить в очередь на передачу) порцию данных, заданную в структуре UserRequest.Io.Tx, в указанный логический канал;
- ***TAU32_Rx_Start*** – запустить приём по указанному логическому каналу. Изменения в конфигурации и состоянии логического канала будут актуализированы при подаче команды TAU32_Configure_Commit;
- ***TAU32_Rx_Stop*** – остановить приём по указанному логическому каналу. Изменения в конфигурации и состоянии логического канала будут актуализированы при подаче команды TAU32_Configure_Commit;
- ***TAU32_Rx_Data*** – принять (поставить буфер в очередь на приём) порцию данных, заданную в структуре UserRequest.Io.Rx, из указанного логического канала;
- ***TAU32_Configure_Channel*** – сконфигурировать указанный логический канал. Изменения в конфигурации и состоянии логического канала будут актуализированы при подаче команды TAU32_Configure_Commit;
- ***TAU32_Timeslots_Complete*** – установить привязку канальных интервалов к логическим каналам, в массиве UserRequest.TimeslotsAssignment.Complete указывается полная информация о привязке. Изменения в конфигурации и состоянии логического канала будут актуализированы при подаче команды TAU32_Configure_Commit;
- ***TAU32_Timeslots_Map*** – установить привязку канальных интервалов к логическим каналам, основываясь на значениях массива UserRequest.TimeslotsAssignment.Map. Каждый элемент массива соответствует логическому каналу, а значение задаёт маску привязываемых канальных интервалов. Изменения в конфигурации и состоянии логического канала будут актуализированы при подаче команды TAU32_Configure_Commit;
- ***TAU32_Timeslots_Channel*** – установить привязку канальных интервалов к логическим каналам, в поле UserRequest.TimeslotsAssignment.Mask указывается битовая маска канальных интервалов, которые необходимо связать с указанным логическим каналом. Изменения в конфигурации и состоянии логического канала будут актуализированы при подаче команды TAU32_Configure_Commit;
- ***TAU32_Configure_Commit*** – запускает актуализацию сделанных изменений в конфигурации;
- ***TAU32_Tx_FrameEnd*** – может использоваться только в комбинации с командой TAU32_Tx_Data, указывая при этом, что необходимо завершить формирование кадра (фрейма) установленного канального протокола;

- ***TAU32_Tx_NoCrc*** – может использоваться только в комбинации с командой TAU32_Tx_Data и только для канального протокола HDLC, указывая, что при передаче необходимо подавить генерацию CRC (вычисление и вставку в передаваемые данные);
- ***TAU32_Configure_E1*** – установить конфигурацию интерфейса E1. В отличие от других команд изменения конфигурации, изменения актуализируются немедленно, использование TAU32_Configure_Commit значения не имеет;

За дополнительной информацией обращайтесь к описанию структуры TAU32_UserRequest{} и к разделам «Обработка запросов» и «TAU32_SubmitRequest()».

TAU32_CHANNEL_CONFIG_BITS

```
#define TAU32_channel_mode_mask     0x…

#define TAU32_data_inversion        0x…
#define TAU32_fr_rx_splitcheck      0x…
#define TAU32_fr_rx_fitcheck        0x…
#define TAU32_fr_tx_auto            0x…
#define TAU32_hdlc_crc32            0x…
#define TAU32_hdlc_adjustment       0x…
#define TAU32_hdlc_interframe_fill  0x…
#define TAU32_hdlc_nocrc            0x…
#define TAU32_tma_flag_filtering    0x…
#define TAU32_tma_flag_nopack       0x…
#define TAU32_tma_flags_mask        0x…
```

Набор констант, используемый (при подаче команды TAU32_Configure_Channel) для указания необходимой конфигурации логических каналов:

- ***TAU32_channel_mode_mask*** – маска, покрывающая биты, используемые для выбора канального протокола (см. «TAU32_CHANNEL_MODES»);
- ***TAU32_data_inversion*** – включает инверсию значений битов данных, как при приёме, так и на передачу;
- ***TAU32_fr_rx_splitcheck*** – включает контроль разбиения принимаемого пакета (фрейма канального протокола) между двумя запросами на приём данных. В случае обнаружения генерируется ошибка TAU32_ERROR_RX_SPLIT (см. «TAU32_ERRORS»);
- ***TAU32_fr_rx_fitcheck*** – включает контроль достаточности места под принимаемый пакет (фрейма канального протокола) в буфере запроса на приём данных. В случае обнаружения генерируется ошибка TAU32_ERROR_RX_UNFIT (см. «TAU32_ERRORS»);
- ***TAU32_fr_tx_auto*** – включает режим, при котором каждый запрос на передачу данных считается отдельным пакетом (фрейма канального протокола),

и нет необходимости указывать TAU32_Tx_FrameEnd (см. «TAU32_REQUEST_COMMANDS»);

- ***TAU32_hdlc_crc32*** – только для HDLC, задает CRC32 (32 бита) для (Cyclic Redundancy Check), иначе используется CRC16 (16 бит);
- ***TAU32_hdlc_adjustment*** – только для HDLC, при включении передатчик будет стараться поддерживать постоянную скорость передачи полезной нагрузки, за счет уменьшения межпакетных интервалов пропорционально количеству бит добавленных при формировании кадров HDLC;
- ***TAU32_hdlc_interframe_fill*** – только для HDLC, задает 0xFF для заполнения межпакетных интервалов, иначе используется 0x7E;
- ***TAU32_hdlc_nocrc*** – только для HDLC, отключает CRC (Cyclic Redundancy Check). При передаче фреймов поле CRC не вычисляется и не передается, при приёме не выделяется и не контролируется;
- ***TAU32_tma_flag_nopack*** – только для прозрачного режима, имеет смысл, только в случае если канальных интервалах, привязанных к логическому каналу, выбраны не все биты: TxFillmask != 0xFF или RxFillmask != 0xFF (см. TAU32_TimeslotAssignment).
  Если указано, то при передаче: каждый байт из буферов маскируется с маской передачи TxFillmask, и целиком помещается в канальный интервал. При приёме все 8 бит канального интервала записываются в память, независимо от значения RxFillmask. Т.е. можно сказать, что с канальным интервалом происходит только «байтовый» обмен.
  Если не указано, то при передаче: из байтов буферов образуется поток бит, который передается только в позициях канального интервала указанных битами TxFillmask. При приёме, из битов, принятых в позициях указанных RxFillmask, формируются октеты (байты) и последовательно записываются в буфера приёма. Т.е. можно сказать, что с канальным интервалом происходит только «битовый» обмен, и только в позициях заданных в TxFillmask/RxFillmask;
- ***TAU32_tma_flag_filtering*** – только для прозрачного режима. Указывает, что приёмопередатчик будет производить фильтрацию принимаемых данных и заполнение «пауз» между запросами на передачу. Для этого используется байт флагов, задаваемый в битах TAU32_tma_flags_mask (см. ниже). При приёме, приёмник будет удалять из принятого потока данных все октеты (байты) со значением байта флагов. При передаче, передатчик будет заполнять паузы между запросами байтом флагов, а также вставлять по одному байту флагов между «пакетами» (запросами с TAU32_Tx_FrameEnd);
- ***TAU32_tma_flags_mask*** – маска, покрывающая биты, используемые для указания байта флагов;
- ***TAU32_tma_flags_shift*** – сдвиг первого бита байта флагов от нулевого бита конфигурации;

Для выбора канального протокола к логическому объединению соответствующих флагов необходимо добавить одну из констант «TAU32_CHANNEL_MODES».

TAU32_CHANNEL_MODES

```
#define TAU32_HDLC              0x…
#define TAU32_V110_x30          0x…
#define TAU32_TMA               0x…
#define TAU32_TMB               0x…
#define TAU32_TMR               0x…
```

Набор констант, определяющий допустимые для логических каналов канальные протоколы:

- ***TAU32_HDLC*** – приём и передача данных осуществляется в пакетном режиме, в соответствии со вторым уровнем протокола HDLC;
- ***TAU32_TMA*** – прозрачный режим, приём и передача данных осуществляется прозрачно, либо с минимальным преобразованием (см. «TAU32_CHANNEL_CONFIG_BITS»);
- ***TAU32_V110_x30*** – V.110 или X.30;
- ***TAU32_TMB*** – режим «B»;
- ***TAU32_TMR*** – режим «R»;

Используется совместно с константами из набора TAU32_CHANNEL_CONFIG_BITS при подаче команды TAU32_Configure_Channel.

В текущей версии DDK полноценно реализована поддержка протокола HDLC и прозрачного режима. За более подробной информацией рекомендуется обращаться к «MUNICH32X Data Sheet», разделу 4 «Detailed Protocol Description».

TAU32_INTERFACE_CONFIG_BITS

```
#define TAU32_line_mode_mask        0x…
#define TAU32_LineOff               0x…
#define TAU32_LineLoopInt           0x…
#define TAU32_LineLoopExt           0x…
#define TAU32_LineNormal            0x…
#define TAU32_LineAIS               0x…

#define TAU32_framing_mode_mask     0x…
#define TAU32_unframed_64           0x…
#define TAU32_unframed_128          0x…
#define TAU32_unframed_256          0x…
#define TAU32_unframed_512          0x…
#define TAU32_unframed_1024         0x…
#define TAU32_unframed_2048         0x…
#define TAU32_unframed              TAU32_unframed_2048
#define TAU32_framed_no_cas         0x…
```

```
#define TAU32_framed_cas_set          0x…
#define TAU32_framed_cas_cross        0x…

#define TAU32_monitor                 0x…
#define TAU32_higain                  0x…
#define TAU32_sa_bypass               0x…
#define TAU32_si_bypass               0x…
#define TAU32_cas_fe                  0x…
#define TAU32_ais_on_loss             0x…
#define TAU32_cas_all_ones            0x…
#define TAU32_cas_io                  0x…
#define TAU32_fas_io                  0x…
#define TAU32_fas8_io                 0x…
#define TAU32_auto_ais                0x…
#define TAU32_not_auto_ra             0x…
#define TAU32_not_auto_dmra           0x…
#define TAU32_ra                      0x…
#define TAU32_dmra                    0x…
#define TAU32_scrambler               0x…
#define TAU32_tx_ami                  0x…
#define TAU32_rx_ami                  0x…
#define TAU32_ja_tx                   0x…

#define TAU32_crc4_mf_tx_only         0x…
#define TAU32_crc4_mf_rx_only         0x…
#define TAU32_crc4_mf \
 (TAU32_crc4_mf_tx_only | TAU32_crc4_mf_rx_only)
```

Набор констант, используемый (при подаче команды TAU32_Configure_E1) для указания необходимой конфигурации интерфейсов E1:

- ***TAU32_line_mode_mask*** – маска, покрывающая биты, используемые для указания режима работы линии E1;
- ***TAU32_LineOff*** – выключить линию E1;
- ***TAU32_LineLoopInt*** – включить внутренний шлейф в передатчике E1. Приёмником будет приниматься то, что передается в линию E1;
- ***TAU32_LineLoopExt*** – включить внешний шлейф в приёмнике E1. Передатчиком будет передаваться то, что принимает приёмник из линии.
- ***TAU32_LineNormal*** – нормальный режим работы;
- ***TAU32_LineAIS*** – включить режим генерации AIS (постоянная передача единиц);
- ***TAU32_framing_mode_mask*** – маска, покрывающая биты, используемые для указания режима структурирования E1;

- ***TAU32_unframed_64*** – неструктурированный режим с занижением скорости до 64 Кбит в секунду, фреймовая структура и цикловая синхронизация по G.704 отсутствуют;
- ***TAU32_unframed_128*** – неструктурированный режим с занижением скорости до 128 Кбит в секунду, фреймовая структура и цикловая синхронизация по G.704 отсутствуют;
- ***TAU32_unframed_256*** – неструктурированный режим с занижением скорости до 256 Кбит в секунду, фреймовая структура и цикловая синхронизация по G.704 отсутствуют;
- ***TAU32_unframed_512*** – неструктурированный режим с занижением скорости до 512 Кбит в секунду, фреймовая структура и цикловая синхронизация по G.704 отсутствуют;
- ***TAU32_unframed_1024*** – неструктурированный режим с занижением скорости до 1024 Кбит в секунду, фреймовая структура и цикловая синхронизация по G.704 отсутствуют;
- ***TAU32_unframed_2048*** – неструктурированный режим без занижения скорости (2048 Кбит в секунду), фреймовая структура и цикловая синхронизация по G.704 отсутствуют;
- ***TAU32_unframed*** – синоним TAU32_unframed_2048 (см. выше);
- ***TAU32_framed_no_cas*** – структурированный режим (фреймовая структура и цикловая синхронизация по G.704 включены), но нет сигнализации CAS и сверхцикловой синхронизации, 16ый канальный интервал интерфейса E1 может быть использован под передачу данных (в том числе CCS);
- ***TAU32_framed_cas_set*** – сигнализация CAS и сверхцикловая синхронизация есть (формируется для передачи и контролируется при приёме), 16ый канальный интервал интерфейса E1 задействуется для обмена сигнализацией через линию E1. Для получения принимаемой сигнализации, и/или формирования сигнализации для передачи, отличной от установленной по умолчанию (0xD), необходимо задействовать низкоуровневый режим побайтового обмена «CAS». Включение блока обслуживания CAS и использование высокоуровневого обмена не имеет смысла (возможно только для чтения получаемой сигнализации);
- ***TAU32_framed_cas_pass*** – сигнализация CAS и сверхцикловая синхронизация есть (не замещается при передаче, но контролируется по приёму). Использование блока обслуживания CAS не возможно. Для получения принимаемой сигнализации, и/или формирования сигнализации для передачи необходимо производить высокоуровневый обмен сигнализацией через канальный интервал приёмопередатчика, который будет соответствовать 16^{му} канальному интервалу линейного интерфейса E1 после прохождения кросс-коннектора;

- ***TAU32_framed_cas_cross*** – сигнализация CAS и сверхцикловая синхронизация есть (формируется для передачи и контролируется при приёме) , 16ый канальный интервал интерфейса E1 задействуется для обмена сигнализацией через линию E1, при этом включена шина обмена сигнализацией между интерфейсом E1 и кросс-коннектором. Для передачи в линию правильных значений сигнализации, необходимо задействовать блок обслуживания CAS (см. TAU32_SetCrossMatrixCas()), и производить высокоуровневый обмен сигнализацией через 16ый канальный интервал приёмопередатчика;
- ***TAU32_monitor*** – на интерфейсе E1 включается режим высокой (линейной) чувствительности приёмника. При включенном режиме происходит линейное усиление входного сигнала. Может использоваться для прослушивания линии, при подключении к ней через резисторы (для снижения нагрузки на передающую сторону);
- ***TAU32_higain*** – на интерфейсе E1 включается режим высокой (нелинейной) чувствительности приёмника. При включенном режиме происходит нелинейное усиление входного сигнала исходя из затухания и искажения сигнала при прохождении по кабелю
- ***TAU32_cas_fe*** – только для структурированного режима E1 с задействованной сверхцикловой синхронизацией CAS. Разрешает «заморозку» последних успешно принятых значений сигнализации CAS при нарушении сверхцикловой синхронизации;
- ***TAU32_ais_on_loss*** – включает режим автоматической передачи AIS (всех неструктурированных единиц) при обнаружении приёмником одной или нескольких из следующих ситуаций: потери несущей (TAU32_RCL), потери кадровой синхронизации (TAU32_RFAS). В случае приёма сигнала AIS (всех неструктурированных единиц, TAU32_RUA1) передача AIS не включается;
- ***TAU32_cas_all_ones*** – только для структурированного режима E1 с задействованной сверхцикловой синхронизацией CAS. Включает режим передачи всех единиц в 16ом канальном интервале;
- ***TAU32_cas_io*** – только для структурированного режима E1 с задействованной сверхцикловой синхронизацией CAS. Включает режим низкоуровневого побайтного обмена данными сигнализации CAS. Обновление информации происходит блоками по 16 байт на границе каждого сверхцикла CAS, поэтому адаптер будет генерировать аппаратные прерывания каждые 2 ms. Непосредственно для обмена данными необходимо использовать функции TAU32_FifoPutCasAppend(), TAU32_FifoPutCasAhead(), TAU32_FifoGetCas();
- ***TAU32_fas_io*** – только для структурированного режима E1. Включает режим низкоуровневого побайтного обмена данными 0го канального интервала E1 (битами Sa4..Sa8, Ra, Si). Обновление информации происходит побайтно на границе каждого нечетного (NAF) фрейма E1, поэтому адаптер будет ге-

нерировать аппаратные прерывания каждые 250 μs. Непосредственно для обмена данными необходимо использовать функции TAU32_FifoPutFasAppend(), TAU32_FifoPutFasAhead(), TAU32_FifoGetFas();

- ***TAU32_fas8_io*** – только для структурированного режима E1. Включает режим низкоуровневого побайтного обмена данными 0ro канального интервала E1 (битами Sa4..Sa8, Ra, Si). Обновление информации происходит блоками по 8 байт (по байту для каждого бита) на границе каждого 8ro нечетного (NAF) фрейма E1, поэтому адаптер будет генерировать аппаратные прерывания каждые 250 μs. Непосредственно для обмена данными необходимо использовать функции TAU32_FifoPutFasAppend(), TAU32_FifoPutFasAhead(), TAU32_FifoGetFas();
- ***TAU32_auto_ais*** – включает режим автоматической передачи AIS (всех неструктурированных единиц), в случае если все канальные интервалы, направленные на передачу в линию E1, имеют аварийное состояние (см. «Неявные действия обслуживания E1»);
- ***TAU32_not_auto_ra*** – только для структурированного режима E1. Выключает автоматическую генерацию сигнала Remote Alarm (бит Ra нулевого канального интервала);
- ***TAU32_not_auto_dmra*** – только для структурированного режима E1 с задействованной сверхцикловой синхронизацией CAS. Выключает автоматическую генерацию сигнала Distant Multiframe Remote Alarm;
- ***TAU32_ra*** – только для структурированного режима E1. Явно включает отправку сигнала Remote Alarm (бит Ra нулевого канального интервала);
- ***TAU32_dmra*** – только для структурированного режима E1 с задействованной сверхцикловой синхронизацией CAS. Явно включает отправку сигнала Distant Multiframe Remote Alarm;
- ***TAU32_scrambler*** – только для неструктурированных режимов, включает скремблер (совместимо с другим оборудование Кроникс);
- ***TAU32_tx_ami*** – при передаче в линию использовать кодирование AMI вместо HDB3;
- ***TAU32_rx_ami*** – при приёме из линии использовать кодирование AMI вместо HDB3;
- ***TAU32_ja_tx*** – включить подавитель фазового дрожания (jitter attenuator) в передающий тракт вместо приёмного. Это может потребоваться при использовании управляемого генератора;
- ***TAU32_crc4_mf*** – только для структурированного режима E1. На интерфейсе задействуется режим сверхцикловой синхронизации контроля CRC4. Производится корректное формирование битов CRC4 (бит Si 0ro канального интервала) при передаче, и контроль при приёме. Если TAU32_crc4_mf выключено (и не задействованы другие режимы), то при передаче Si бит уста-

навливается равным «1», а при приёме игнорируется. Является объединением режимов TAU32_crc4_mf_rx_only и TAU32_crc4_mf_tx_only;

- ***TAU32_crc4_mf_rx_only*** – только для структурированного режима E1. Производится контроль CRC4 (бит Si 0го канального интервала) при приёме. Индикация ошибок в E-битах (передаваемых в линию E1) производится, только если также задействован режим TAU32_crc4_mf_**tx**_only;
- ***TAU32_crc4_mf_tx_only*** – только для структурированного режима E1. Производится корректное формирование битов CRC4 (бит Si 0го канального интервала) при передаче. Контроль CRC4 и индикация ошибок в E-битах производится, только если также задействован режим TAU32_crc4_mf_**rx**_only;
- ***TAU32_sa_bypass*** – только для структурированного режима E1. Бит Sa в 0ом канальном интервале будет прозрачно транслироваться между интерфейсами E1 и кросс-коннектором;
- ***TAU32_si_bypass*** – только для структурированного режима E1. Биты Si в 0ом канальном интервале будут прозрачно транслироваться между интерфейсами E1 и кросс-коннектором;

TAU32_ERRORS

```
#define TAU32_NOERROR               0x…
#define TAU32_SUCCESSFUL            0x…
#define TAU32_ERROR_ALLOCATION      0x…
#define TAU32_ERROR_BUS             0x…
#define TAU32_ERROR_FAIL            0x…
#define TAU32_ERROR_TIMEOUT         0x…
#define TAU32_ERROR_CANCELLED       0x…

#define TAU32_ERROR_TX_UNDERFLOW    0x…
#define TAU32_ERROR_TX_PROTOCOL     0x…

#define TAU32_ERROR_RX_OVERFLOW     0x…
#define TAU32_ERROR_RX_ABORT        0x…
#define TAU32_ERROR_RX_CRC          0x…
#define TAU32_ERROR_RX_SHORT        0x…
#define TAU32_ERROR_RX_SYNC         0x…
#define TAU32_ERROR_RX_FRAME        0x…
#define TAU32_ERROR_RX_LONG         0x…
#define TAU32_ERROR_RX_SPLIT        0x…
#define TAU32_ERROR_RX_UNFIT        0x…

#define TAU32_ERROR_INT_OVER_TX     0x…
#define TAU32_ERROR_INT_OVER_RX     0x…
#define TAU32_ERROR_INT_STORM       0x…
#define TAU32_ERROR_INT_E1LOST      0x…
```

Набор констант, определяющий битовые флаги возможных ошибок приёма, передачи и управления:

- ***TAU32_NOERROR*** – нет ошибок;

- ***TAU32_SUCCESSFUL*** – синоним TAU32_NOERROR;
- ***TAU32_ERROR_ALLOCATION*** – исчерпан пул внутренних структур. DDK позволяет ставить в очередь до 512 запросов, 256 из которых могут быть запросами на приём или передачу;
- ***TAU32_ERROR_BUS*** – ошибка доступа (таймаут) к шине PCI или доступа к данным в оперативной памяти через шину PCI. Может возникать в случае, когда передача запускается одновременно по нескольким логическим каналам, вследствие того, что адаптеру не удается вовремя считать данные через PCI-шину для заполнения нескольких внутренних FIFO одновременно;
- ***TAU32_ERROR_FAIL*** – фатальная ошибка, команда управления логическими каналами завершилась неудачей;
- ***TAU32_ERROR_TIMEOUT*** – фатальная ошибка, (таймаут) при выполнении команды управления логическими каналами;
- ***TAU32_ERROR_CANCELLED*** – запрос был отменен;
- ***TAU32_ERROR_TX_UNDERFLOW*** – для передачи не был вовремя предоставлен очередной буфер (запрос на передачу) с данными;
- ***TAU32_ERROR_TX_PROTOCOL*** – ошибка канального протокола, например вследствие остановки передатчика или отмены запроса;
- ***TAU32_ERROR_RX_OVERFLOW*** – для приёма данных вовремя не был предоставлен очередной буфер (запрос на приём);
- ***TAU32_ERROR_RX_ABORT*** – приём пакета (фрейма канального протокола) был прерван, например вследствие отмены запроса или остановки приёмника;
- ***TAU32_ERROR_RX_CRC*** – ошибка контрольной суммы HDLC;
- ***TAU32_ERROR_RX_SHORT*** – принят слишком короткий пакет (фрейм канального протокола), например фрейм HDLC короче длины контрольной суммы;
- ***TAU32_ERROR_RX_SYNC*** – три (или более) фрейма подряд приняты с ошибкой в синхронизационной последовательности;
- ***TAU32_ERROR_RX_FRAME*** – ошибка канального протокола при приёме пакета (фрейма), например длина принятого кадра HDLC не кратна 8;
- ***TAU32_ERROR_RX_LONG*** – приёмник обнаружил слишком длинный пакет (фрейм канального протокола);
- ***TAU32_ERROR_RX_SPLIT*** – задействована опция TAU32_fr_rx_splitcheck, и в буфере запроса находится более одного пакета (фрейма канального протокола);
- ***TAU32_ERROR_RX_UNFIT*** – задействована опция TAU32_fr_rx_fitcheck, и принятый пакет (фрейма канального протокола) не поместился в буфер одного запроса;

- ***TAU32_ERROR_INT_OVER_TX*** – фатальная ошибка, переполнение очереди прерываний от передатчика;
- ***TAU32_ERROR_INT_OVER_RX*** – фатальная ошибка, переполнение очереди прерываний от приёмника;
- ***TAU32_ERROR_INT_STORM*** – фатальная ошибка, чрезвычайно много аппаратных прерываний (≈ 1000 на один вызов обработчика);
- ***TAU32_ERROR_INT_E1LOST*** – вследствие задержек аппаратное прерывание от интерфейса E1 не было обработано вовремя. Если задействован один из низкоуровневых режимов обмена, то скорее всего порция данных была потеряна (см. «Обслуживание прерываний»);

При обнаружении ошибок DDK уведомляет функцию обратного вызова TAU32_UserContext.pErrorNotifyCallback().

Для получения дополнительной информации смотрите раздел «Обратные вызовы» и «TAU32_UserContext{}».

TAU32_STATUS

```
#define TAU32_FRLOMF         0x…
#define TAU32_CROSS_WAITING  0x…
#define TAU32_CROSS_PENDING  0x…
#define TAU32_LED            0x…
```

Набор констант, определяющий битовые флаги статуса адаптера:

- ***TAU32_FRLOMF*** – блок обслуживания сигнализации CAS включен и потерял синхронизацию (ведет её поиск) при приёме 16^{ти} байтовых блоков из 16го канального интервала;
- ***TAU32_CROSS_PENDING*** – кросс-коннектор ожидает границы фрейма E1 для обновления таблицы кросс-коммутации;
- ***TAU32_CROSS_WAITING*** – DDK ожидает, когда кросс-коннектор обновит таблицу кросс-коммутации, с тем чтобы обновить её еще раз. Такая ситуация возникает когда таблица кросс-коммутации изменяется вами чаще, чем один раз в 125 μs;
- ***TAU32_LED*** – отражает состояние светодиода адаптера;

При изменении статуса адаптера DDK уведомляет функцию обратного вызова TAU32_UserContext.pStatusNotifyCallback() по следующим правилам:

- При любом изменении флага TAU32_FRLOMF;
- При деактивации флагов TAU32_CROSS_PENDING или TAU32_CROSS_WAITING;
- О других изменениях уведомление не производится;

Для получения дополнительной информации смотрите раздел «Обратные вызовы».

TAU32_INTERFACES

```
#define TAU32_E1_ALL            0x…
#define TAU32_E1_A              0x…
#define TAU32_E1_B              0x…
```

Набор констант, используемый для ссылки на интерфейсы E1 при вызове функций DDK:

- ***TAU32_E1_ALL*** – оба интерфейса;
- ***TAU32_E1_A*** – только первый (E1/0) интерфейс;
- ***TAU32_E1_B*** – только второй (E1/1) интерфейс;

Не во всех случаях допустимо ссылаться на оба интерфейса сразу, смотрите описание соответствующих функций и их параметров.

TAU32_LIMITS

```
#define TAU32_CHANNELS          32
#define TAU32_TIMESLOTS         32
#define TAU32_MAX_INTERFACES    2
#define TAU32_MTU               8184
#define TAU32_IO_QUEUE          4
#define TAU32_MAX_REQUESTS      512
#define TAU32_MAX_BUFFERS       256
#define TAU32_FIFO_SIZE         256
```

Набор констант, определяющих важные параметры и ограничения конкретной версии DDK.

- ***TAU32_CHANNELS*** – максимально допустимое (обслуживаемое) количество логических каналов в данной версии DDK;
- ***TAU32_TIMESLOTS*** – максимально допустимое (обслуживаемое) количество канальных интервалов приёмопередатчика в данной версии DDK;
- ***TAU32_MAX_INTERFACES*** – максимальное количество интерфейсов E1 на одном адаптере Tau32;
- ***TAU32_MTU*** – максимальный размер порции данных, которая может быть принята или передана одним запросом;
- ***TAU32_IO_QUEUE*** – минимальный размер каждой из очередей запросов на приём и передачу, для обеспечения «гладкого» обмена данными без остановок и обрывов;
- ***TAU32_MAX_REQUESTS*** – максимальное количество одновременно всех обслуживаемых и поставленных в очередь запросов;
- ***TAU32_MAX_BUFFERS*** – максимальное количество одновременно обслуживаемых и поставленных в очередь запросов на приём или передачу данных;

- ***TAU32_FIFO_SIZE*** – размер FIFO буферов низкоуровневого обмена;

TAU32_SYNC_MODES

```
#define TAU32_SYNC_INTERNAL     0x…
#define TAU32_SYNC_RCV_A        0x…
#define TAU32_SYNC_RCV_B        0x…
```

Набор констант, определяющих возможные режимы (источники) синхронизации адаптера (см. «TAU32_SetSyncMode()»):

- ***TAU32_SYNC_INTERNAL*** – использовать внутренний генератор в качестве источника единой синхронизации;
- ***TAU32_SYNC_RCV_A*** – использовать частоту, восстановленную приёмником **первого** (E1/**0**) интерфейса, в качестве источника единой синхронизации;
- ***TAU32_SYNC_RCV_B*** – использовать частоту, восстановленную приёмником **второго** (E1/**1**) интерфейса, в качестве источника единой синхронизации;
- ***TAU32_SYNC_LYGEN*** – использовать внутренний управляемый генератор, перед установкой управляемый генератор должен быть настроен;

Если в режимах TAU32_SYNC_RCV_A и TAU32_SYNC_RCV_B источник синхронизации будет потерян, либо выйдет за допустимые пределы, то адаптер автоматически перейдёт на синхронизацию от внутреннего генератора.

TAU32_INTERFACE_STATUS_BITS

```
#define TAU32_RCL        0x…
#define TAU32_RLOS       0x…
#define TAU32_RUA1       0x…
#define TAU32_RRA        0x…
#define TAU32_RSA1       0x…
#define TAU32_RSA0       0x…
#define TAU32_RDMA       0x…
#define TAU32_LOTC       0x…
#define TAU32_RSLIP      0x…
#define TAU32_TSLIP      0x…
#define TAU32_RFAS       0x…
#define TAU32_RCRC4      0x…
#define TAU32_RCAS       0x…
#define TAU32_RJITTER    0x…
#define TAU32_RCRC4LONG  0x…
#define TAU32_E1OFF      0x…
#define TAU32_LOS        TAU32_RLOS
#define TAU32_AIS        TAU32_RUA1
#define TAU32_LOF        TAU32_RFAS
#define TAU32_AIS16      TAU32_RSA1
#define TAU32_LOMF       TAU32_RCAS
#define TAU32_FLOMF      TAU32_RDMA
```

Набор констант, определяющий битовые флаги статуса интерфейсов E1:

- ***TAU32_RCL*** – (receive carrier lost) потеря несущей. Если выбран HDB3 код, то в линии E1 отсутствует сигнал. Если выбран AMI код, то либо в линии E1 отсутствует сигнал, либо принимается неструктурированный поток нулевых бит;
- ***TAU32_RLOS*** – (receive sync lost) приёмник не синхронизирован с потоком E1/ИКМ-30 (комбинация TAU32_RCL или TAU32_RFAS или TAU32_RCRC4 или TAU32_RCAS);
- ***TAU32_RUA1*** – (received unframed all ones) принимается неструктурированный поток единичных бит (сигнал AIS);
- ***TAU32_RRA*** – (receive remote alarm) принят сигнал удаленной аварии;
- ***TAU32_RSA1*** – (receive signaling all ones) в 16ом канальном интервале принимаются все единицы;
- ***TAU32_RSA0*** – (receive signaling all zeros) в 16ом канальном интервале принимаются все нули;
- ***TAU32_RDMA*** – (receive distant multiframe alarm) принимается сигнал удаленной тревоги;
- ***TAU32_LOTC*** – (transmit clock lost) передатчик не получает синхросигнал;

- ***TAU32_RSLIP*** – (receiver slip event) из-за различия частоты синхронизации в FIFO приёмника был удален или повторен один фрейм E1;
- ***TAU32_TSLIP*** – (transmitter slip event) из-за различия частоты синхронизации в FIFO передатчика был удален или повторен один фрейм E1;
- ***TAU32_RFAS*** – (receiver lost and searching for FAS) приёмник потерял синхронизацию с цикловой структурой (FAS), и производит её поиск;
- ***TAU32_RCRC4*** – (receiver lost and searching for CRC4 MF) приёмник потерял синхронизацию со сверхцикловой структурой для CRC4, и производит её поиск;
- ***TAU32_RCAS*** – (received lost and searching for CAS MF) приёмник потерял синхронизацию со сверхцикловой структурой для CAS, и производит её поиск;
- ***TAU32_RJITTER*** – (jitter attenuator limit) уровень FIFO подавления фазового дрожания близок к границе;
- ***TAU32_RCRC4LONG*** – (G.706 400ms limit of searching for CRC4) синхронизация со сверхцикловой структурой для CRC4 была потеряна и не найдена в течении 400 ms;
- ***TAU32_E1OFF*** – (E1 line power-off) интерфейс выключен;
- ***TAU32_LOS*** – синоним TAU32_RLOS;
- ***TAU32_AIS*** – синоним TAU32_RUA1;
- ***TAU32_LOF*** – синоним TAU32_RFAS;
- ***TAU32_AIS16*** – синоним TAU32_RSA1;
- ***TAU32_LOMF*** – синоним TAU32_RCAS;
- ***TAU32_FLOMF*** – синоним TAU32_RDMA;

Все аварийные ситуации обнаруживаются, только если задействованы соответствующие режимы работы интерфейса E1.

При изменении статуса интерфейсов E1 DDK уведомляет функцию обратного вызова TAU32_UserContext.pStatusNotifyCallback().

Для получения дополнительной информации смотрите раздел «Обратные вызовы» и «Критерии аварийных ситуаций E1».

TAU32_FIFO_ID

```
#define TAU32_FifoId_CasRx 0x…
#define TAU32_FifoId_CasTx 0x…
#define TAU32_FifoId_FasRx 0x…
#define TAU32_FifoId_FasTx 0x…

#define TAU32_FifoId_Max   0x…
```

Набор констант, идентифицирующих FIFO-буфера, связанные с каждым из интерфейсов E1:

- ***TAU32_FifoId_CasRx*** – константа-идентификатор FIFO приёма данных сигнализации CAS;
- ***TAU32_FifoId_CasTx*** – константа-идентификатор FIFO передачи данных сигнализации CAS;
- ***TAU32_FifoId_FasRx*** – константа-идентификатор FIFO приёма значений битов 0^{ro} канального интервала (Sa4..Sa8, Ra, …);
- ***TAU32_FifoId_FasTx*** – константа-идентификатор FIFO передачи значений битов 0^{ro} канального интервала (Sa4..Sa8, Ra, …);

За дополнительной информацией обращайтесь к разделам «Низкоуровневый обмен» и «TAU32_SetFifoTrigger()».

TAU32_E1_State{}

```
typedef struct tag_TAU32_E1_State
{
    unsigned __int32 TickCounter;
    unsigned __int32 RxViolations;
    unsigned __int32 Crc4Errors;
    unsigned __int32 FarEndBlockErrors;
    unsigned __int32 FasErrors;
    unsigned __int32 TransmitSlips;
    unsigned __int32 ReceiveSlips;
    unsigned __int32 Status;
    unsigned __int32 FifoSlip[TAU32_FifoId_Max];
} TAU32_E1_State;
```

Структура, хранящая информацию об интерфейсе E1, его статус и счетчики ошибок:

- ***TickCounter*** – счетчик тиков 1/16 секунды;
- ***RxViolations*** – счетчик нарушений кодирования. Считаются ситуации «Code Violations» если приёмник настроен на код HDB3, или «Bipolar Violations» для кода AMI;
- ***Crc4Errors*** – счетчик ошибок CRC4, счетчик изменяется только когда включен режим CRC4 и приёмник удерживает сверхцикловую синхронизацию для CRC4;

- ***FarEndBlockErrors*** – счетчик ошибок CRC4 индицированных удаленной стороной (посредством значений E-битов), счетчик изменяется только когда включен режим CRC4 и приёмник удерживает сверхцикловую синхронизацию для CRC4;
- ***FasErrors*** – счетчик ошибок нарушений цикловой синхронизации (FAS), счетчик изменяется только когда включен структурированный режим и приёмник удерживает цикловую синхронизацию;
- ***TransmitSlips*** – счетчик ситуаций когда из-за различия частоты синхронизации в FIFO передатчика удаляется или повторяется фрейм E1;
- ***ReceiveSlips*** – счетчик ситуаций когда из-за различия частоты синхронизации в FIFO приёмника удаляется или повторяется фрейм E1;
- ***Status*** – статус интерфейса E1, комбинация битовых флагов TAU32_INTERFACE_STATUS_BITS;
- ***FifoSlip*** – массив счетчиков по одному для каждого FIFO-буферов низкоуровневого обмена. Счетчики считают ситуации, когда соответствующее FIFO либо не может вместить новую порцию принятых данных, либо не содержит достаточно данных для передачи;

Все счетчики, кроме TransmitSlips, ReceiveSlips и FifoSlip обновляются 16 раз в секунду, одновременно с тиками таймера. Счетчики TransmitSlips, ReceiveSlips и FifoSlip инкрементируются непосредственно в момент обнаружения ошибки.

При изменении статуса интерфейса E1 DDK уведомляет функцию обратного вызова TAU32_UserContext.pStatusNotifyCallback().

Для получения дополнительной информации смотрите раздел «Обратные вызовы» и «

Критерии аварийных ситуаций E1».

TAU32_TimeslotAssignment{}

```
typedef struct tag_TAU32_TimeslotAssignment
{
    unsigned __int8 TxChannel, RxChannel;
    unsigned __int8 TxFillmask, RxFillmask;
} TAU32_TimeslotAssignment;
```

Структура, определяющая привязку одного канального интервала приёмопередатчика к логическим каналам:

- ***TxChannel*** – номер логического канала, к передатчику которого привязывается данный канальный интервал. Т.е. канал, при передаче из которого, данные будут поступать в этот (но не только в этот) канальный интервал. Для того чтобы указать, что канальный интервал не должен использоваться для передачи, задайте значение 0xFF;

- ***RxChannel*** – номер логического канала, к приёмнику которого привязывается данный канальный интервал. Т.е. канал, в который для приёма будут направляться данные из этого (но не только из этого) канального интервала. Для того чтобы указать, что канальный интервал не должен использоваться для приёма, задайте значение 0xFF;
- ***TxFillmask*** – битовая маска, определяющая биты канального интервала, которые будут задействованы для передачи информации из логического канала. Для того чтобы указать, что канальный интервал не должен использоваться для передачи, задайте значение 0x00;
- ***RxFillmask*** – битовая маска, определяющая биты канального интервала, информация из которых будет направлена в приёмник логического канала. Для того чтобы указать, что канальный интервал не должен использоваться для приёма, задайте значение 0x00;

Массив из TAU32_TIMESLOTS таких элементов образует поле TimeslotsAssignment.Complete в структуре TAU32_UserRequest. Так задается полная информация о привязке канальных интервалов к логическим каналам, при подаче команды TAU32_Timeslots_Complete (см. «TAU32_UserRequest{}» и «TAU32_REQUEST_COMMANDS»).

При использовании отдельных (не всех) битов в канальном интервале, необходимо помнить, что порядок следования битов в линии E1 определяется настройками кросс-коннектора, см. «Порядок битов в канальных интервалах» и «TAU32_SetCrossMatrix()».

TAU32_CrossMatrix{}

```
#define TAU32_CROSS_WIDTH       96
#define TAU32_CROSS_OFF         0x…

typedef unsigned __int8 TAU32_CrossMatrix[TAU32_CROSS_WIDTH];
```

Массив из ***TAU32_CROSS_WIDTH*** элементов, определяющий матрицу кросс-коммутации. Каждый элемент массива соответствует канальному интервалу «выходящему» из кросс-коннектора, а значение элемента задает «входящий» в кросс-коннектор канальный интервал, данные из которого будут направлены в «выходящий» канальный интервал. Другими словами, для каждого канального интервала на выходе кросс-коннектора задается «источник», которым может быть любой другой канальный интервал.

Элементы матрицы кросс-коммутации образуют три секции (две секции для модели Lite) по 32 элемента. Первая секция (элементы матрицы с 0 по 31) соответствует канальным интервалам приёмопередатчика, вторая секция (элементы матрицы с 32 по 63) соответствует канальным интервалам первого (E1/0) интерфейса E1, третья секция (элементы матрицы с 64 по 95) соответствует канальным интервалам второго (E1/1) интерфейса E1.

Для того чтобы указать, что канальный интервал не должен быть подключен к какому-либо источнику, задайте в соответствующем элементе матрицы значение ***TAU32_CROSS_OFF***.

Для установки матрицы кросс-коммутации вам необходимо сформировать её образ, и передать адрес в функцию «TAU32_SetCrossMatrix()».

Для получения дополнительной информации смотрите раздел «Устройство адаптера» и «TAU32_SetCrossMatrix()».

TAU32_SaCross{}

```
#define TAU32_SaDisable     0x…
#define TAU32_SaSystem      0x…
#define TAU32_SaIntA        0x…
#define TAU32_SaIntB        0x…
#define TAU32_SaAllZeros    0x…

typedef struct tag_TAU32_SaCross
{
    unsigned __int8 InterfaceA, InterfaceB;
    unsigned __int8 SystemEnableTs0;
} TAU32_SaCross;
```

Набор констант и структура для указания конфигурации коммутатора Sa-бит. Для каждого из интерфейсов E1, по принципу кросс-коннектора, задается источник Sa-бит, и отдельно включается обмен с приёмопередатчиком через 0ой канальный интервал.

Элементы структуры InterfaceA и InterfaceB задают источники Sa-бит для первого (E1/**0**) и второго (E1/**1**) интерфейсов соответственно:

- ***TAU32_SaDisable*** – коммутация Sa-битов выключена (передаются все единицы);
- ***TAU32_SaSystem*** – источником является 0ой канальный интервал приёмопередатчика;
- ***TAU32_SaIntA*** – источником является первый (E1/**0**) интерфейс;
- ***TAU32_SaIntB*** – источником является второй (E1/**1**) интерфейс;
- ***TAU32_SaAllZeros*** – во всех Sa-битах передаются нули;

Ненулевое значение элемента структуры SystemEnableTs0 включает передачу Sa-битов от коммутатора к приёмопередатчику. При этом 0ой канальный интервал приёмопередатчика переключается от кросс-коннектора к коммутатору Sa-бит.

Для установки конфигурации коммутатора Sa-битов, вам необходимо сформировать образ структуры TAU32_SaCross, и передать адрес в функцию «TAU32_SetSaCross()». За дополнительной информацией обращайтесь к разделу «Коммутатор Sa-бит».

Функции и макросы

TAU32_Initialize()

```
BOOLEAN TAU32_Initialize(
    TAU32_UserContext *pUserContext,
    BOOLEAN CronyxDiag);
```

Функция TAU32_Initialize() выполняет инициализацию адаптера, его базовое тестирование и в случае успеха инициализирует внутренние структуры DDK.

Параметры:

- pUserContext – указатель на подготовленный экземпляр структуры TAU32_UserContext;
- CronyxDiag – для внутреннего использование Кроникс, вы всегда должны передавать значение «0» (FALSE);

Возвращаемый результат:

- не нулевое (TRUE) значение, если инициализация прошла успешно;

Функция TAU32_Initialize() не должна вызываться повторно, для уже успешно инициализированных адаптеров, но может вызываться повторно после остановки адаптера функцией TAU32_DestructiveHalt().

Выполнение функции TAU32_Initialize() может потребовать до 100 ms времени, поэтому функция должна вызываться на открытых прерываниях и без захвата ресурсов синхронизации. Например, в случае ОС Windows, инициализация должна производиться при IRQL = PASSIVE_LEVEL (см. Windows DDK).

Функция TAU32_Initialize() обновляет некоторые элементы указанной структуры TAU32_UserContext. Если функция вернула значение «0» (FALSE), то никакая другая функция DDK не может использоваться, а элемент pUserContext→InitErrors будет содержать информацию об обнаруженных проблемах и неисправностях.

Выполнение TAU32_Initialize() не вызывает генерацию адаптером аппаратных прерываний, если только оборудование не имеет кардинальных повреждений. После инициализации, генерация аппаратных прерываний адаптером запрещена. Для завершения подготовки DDK и адаптера к работе, необходимо активировать (подключить средствами операционной системы) обработчик аппаратных прерываний DDK (функцию TAU32_HandleInterrupt()) и затем разрешить генерацию адаптером аппаратных прерываний функцией TAU32_EnableInterrupts(). После этого DDK будет полностью готов к работе.

В версии DDK для ОС Windows, функция расположена в PAGEABLE (выгружаемой) секции программного кода. За дополнительной информацией обращайтесь к разделам «Инициализация и запуск» и «TAU32_UserContext{}».

TAU32_DestructiveHalt()

```
void TAU32_DestructiveHalt(
    TAU32_Controller *pControllerObject,
    BOOLEAN CancelRequests);
```

Функция TAU32_DestructiveHalt() производит остановку адаптера и DDK с разрушением внутренних структур DDK.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- CancelRequests – ненулевое значение (TRUE) уведомляет DDK о необходимости вызвать функции обратного вызова для каждого из запросов, которые выполнялись или находились в очереди;

Возвращаемый результат:

- нет (void);

Функция TAU32_DestructiveHalt() должна вызываться только для адаптеров, которые были успешно инициализированы функцией TAU32_Initialize(), и не должна вызываться повторно до завершения успешной новой инициализации.

Выполнение функции TAU32_DestructiveHalt() может потребовать до 100 ms времени, поэтому функция должна вызываться на открытых прерываниях и без захвата ресурсов синхронизации. Например, в случае ОС Windows, инициализация должна производиться при IRQL = PASSIVE_LEVEL (см. Windows DDK).

В случае обнаружения аварийной ситуации, когда функция TAU32_DestructiveHalt() не может быть вызвана, необходимо запретить генерацию адаптером аппаратных прерываний вызовом функции TAU32_DisableInterrupts(), и произвести остановку позже.

Перед выполнением остановки, необходимо запретить генерацию адаптером аппаратных прерываний функцией TAU32_DisableInterrupts(), затем деактивировать (отключить средствами операционной системы) обработчик аппаратных прерываний DDK (функцию TAU32_HandleInterrupt()), и только после этого производить остановку.

После выполнения остановки, никакая функция DDK, кроме TAU32_Initialize(), не должна вызываться для соответствующего адаптера.

В версии DDK для ОС Windows, функция расположена в PAGEABLE (выгружаемой) секции программного кода. За дополнительной информацией обращайтесь к разделам «Остановка» и «TAU32_UserContext{}».

TAU32_BeforeReset()

```
BOOLEAN TAU32_BeforeReset(
    TAU32_UserContext *pUserContext);
```

Функция TAU32_BeforeReset() доступна в DDK начиная с версии 1.2, и предназначена для обхода аппаратной ошибки в контроллере Infineon MUNICH32X, вследствие которой возможна генерация аппаратного прерывания (сигнал INTA шины PCI) при сбросе адаптера.

Параметры:

- pUserContext – указатель на подготовленный экземпляр структуры TAU32_UserContext. Структура должна быть подготовлена также как и ля функции TAU32_Initialize();

Возвращаемый результат:

- нет (void);

Для подавления генерации «неожиданного» сигнала аппаратного прерывания необходимо вызвать TAU32_BeforeReset() непосредственно и только перед сбросом адаптера посредством записи констант TAU32_PCI_RESET_ON и TAU32_PCI_RESET_OFF в регистр TAU32_PCI_RESET_ADDRESS.

В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделу «Идентификаторы адаптера на шине PCI».

TAU32_HandleInterrupt()

```
BOOLEAN TAU32_HandleInterrupt(
    TAU32_Controller *pControllerObject);
```

Эта функция является обработчиком аппаратных прерываний сгенерированных адаптером, может использоваться для обработки прерываний в режиме опроса (polling mode).

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;

Возвращаемый результат:

- не нулевое (TRUE) значение, если на адаптере имелось необработанное аппаратное прерывание;

Функция TAU32_HandleInterrupt() должна быть использована для обработки сгенерированных адаптером аппаратных прерываний. Для этого вам необходимо, во взаимодействии со средствами операционной системы, соответствующим образом подключить функцию TAU32_HandleInterrupt() к обработке прерываний по линии IRQ, которая назначена адаптеру. Для возможности полноценного разделения линии IRQ, функция TAU32_HandleInterrupt() возвращает булевой результат.

Функция TAU32_HandleInterrupt() выполняет массу действий связанных с работой адаптера Tau32, и в процессе работы может неоднократно вызывать различные, указанные вами, функции обратных вызовов.

Другие функции DDK не должны прерываться обработчиком аппаратных прерываний того же адаптера, и поэтому должны быть соответствующим образом защищены (см. KeSynchronizeExecution() в Windows DDK). Функция TAU32_HandleInterrupt() производит контроль такой ситуации, и в случае обнаружения поток выполнения будет остановлен на контрольной отладочной точке (int 3).

В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделам «Обслуживание прерываний» и «Обработка запросов».

TAU32_IsInterruptPending(), Linux SMP

```
BOOLEAN TAU32_IsInterruptPending(
    TAU32_Controller *pControllerObject);
```

Функция TAU32_IsInterruptPending() позволяет быстро проверить, сгенерировано ли адаптером аппаратное прерывание, или нет.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;

Возвращаемый результат:

- не нулевое (TRUE) значение, если на адаптере имеется необработанное аппаратное прерывание;

Функция предназначена, прежде всего, для использования в ОС Linux на SMP (Symmetric Multi Processor) платформах, в обработчике аппаратных прерываний. На SMP платформах в ОС Linux вызов основной функции обработки прерываний TAU32_HandleInterrupt(), должен быть защищен от повторного входа на другом процессоре при помощи spin_lock_irq() и spin_unlock_irq(). Использование TAU32_IsInterruptPending() позволяет обойти последовательность вызовов spin_lock_irq(), TAU32_HandleInterrupt() и spin_unlock_irq(), в случае если на адаптере нет необработанных прерываний.

Функция TAU32_IsInterruptPending() должна вызываться только для адаптеров, которые были успешно инициализированы.

В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделу «Обслуживание прерываний».

TAU32_EnableInterrupts(), TAU32_DisableInterrupts()

```
void TAU32_EnableInterrupts(
    TAU32_Controller *pControllerObject);

void TAU32_DisableInterrupts(
    TAU32_Controller *pControllerObject);
```

Функции TAU32_EnableInterrupts() и TAU32_DisableInterrupts() производят соответственно разрешение и запрещение генерации адаптером аппаратных прерываний.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;

Возвращаемый результат:

- нет (void);

Функции TAU32_EnableInterrupts() и TAU32_DisableInterrupts() должны вызываться только для адаптеров, которые были успешно инициализированы функцией TAU32_Initialize(), и еще не остановлены функцией TAU32_DestructiveHalt().

В версии DDK для ОС Windows, обе функции расположена в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделам «Обслуживание прерываний» и «Обработка запросов».

TAU32_SubmitRequest()

```
BOOLEAN TAU32_SubmitRequest(
    TAU32_Controller *pControllerObject,
    TAU32_UserRequest *pRequest);
```

Функция TAU32_SubmitRequest() производит установку запроса TAU32_UserRequest в очередь на асинхронное выполнение.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- pRequest – указатель на подготовленный экземпляр структуры TAU32_UserRequest;

Возвращаемый результат:

- не нулевое (TRUE) значение, если запрос принят к исполнению;
- нулевое значение (FALSE), означает, что заданы неверные или недопустимые параметры;

Функция TAU32_SubmitRequest() должна вызываться только для адаптеров, которые были успешно инициализированы. При вызове функция должна быть защищена от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

При вызове экземпляр структуры TAU32_UserRequest должен быть правильно заполнен. Если запрос содержит неверный, неполный или некорректный набор параметров, то запрос будет отвергнут, и TAU32_SubmitRequest() вернет значение «0» (FALSE). В противном случае запрос принимается к исполнению и контроль над запросом переходит от вашего кода к DDK, до уведомления функции обратного вызова pRequest→ pCallback(), либо до успешной отмены запроса функцией

TAU32_CancelRequest() или до завершения остановки DDK функцией TAU32_DestructiveHalt().

Пока контроль над запросом находится у DDK, вы не должны изменять переданные параметры, или производить с запросом какие-либо действия кроме попытки его отмены. В процессе работы функция TAU32_SubmitRequest() может неоднократно вызывать различные функции обратных вызовов.

В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделам «TAU32_UserContext{}» и «Обработка запросов».

TAU32_CancelRequest()

```
BOOLEAN TAU32_CancelRequest(
    TAU32_Controller *pControllerObject,
    TAU32_UserRequest *pRequest,
    BOOLEAN BreakIfRunning);
```

Функция TAU32_CancelRequest() производит попытку отмены запроса, который ранее был поставлен в очередь на выполнение функцией TAU32_SubmitRequest().

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- pRequest – указатель на отменяемый запрос;
- BreakIfRunning – ненулевое значение (TRUE), означает, что необходимо отменить запрос, даже если он уже выполняется. В противном случае отмена произойдет, если только запрос все еще ожидает выполнения в очереди;

Возвращаемый результат:

- не нулевое (TRUE) значение, если запрос успешно отменен;
- нулевое значение (FALSE), означает, что запрос не может быть отменен немедленно, или заданы неверные параметры;

Функция TAU32_CancelRequest() должна вызываться только для адаптеров, которые были успешно инициализированы. При вызове функция должна быть защищена от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

Функцию следует вызывать только для запросов, которые ранее были приняты к исполнению функцией TAU32_SubmitRequest(). Если на момент вызова запрос уже был выполнен, то функция не произведет каких-либо действий, и возвратит значение «0» (FALSE).

Если функция способна отменить выполнение запроса непосредственно во время вызова, то запрос будет немедленно отменен, и функция вернет ненулевое значение (TRUE). В противном случае запрос будет помечен для отмены в асинхрон-

ном режиме, функция вернет значение «0» (FALSE), а о завершении асинхронной отмены, ваш код будет уведомлен через функцию обратного вызова.

Если запрос уже начал выполняться, и параметр BreakIfRunning равен «0» (FALSE), функция TAU32_CancelRequest() вернет значение 0 (FALSE) и запрос будет выполняться дальше. Если же при этом BreakIfRunning не равен нулю (TRUE), то DDK прервет выполнение запроса. Однако отмена запросов, которые уже начали выполняться, требует асинхронного выполнения, а также будет порождать дополнительные ошибки приёма/передачи данных и управления логическими каналами.

В процессе работы функция TAU32_SubmitRequest() может неоднократно вызывать различные функции обратных вызовов. В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделам «Обработка запросов» и «TAU32_UserContext{}».

TAU32_LedBlink(), TAU32_LedSet()

```
void TAU32_LedBlink(
    TAU32_Controller *pControllerObject);

void TAU32_LedSet(
    TAU32_Controller *pControllerObject,
    BOOLEAN On);
```

Обе функции предназначены для управления светодиодным индикатором адаптера. Функция TAU32_LedSet() явно зажигает или гасит индикатор в зависимости от второго переданного ей параметра. А функция TAU32_LedBlink() меняет состояние индикатора на противоположное, «мигает» им.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- On (только для TAU32_LedSet()) – ненулевое значение (TRUE) означает, что необходимо включить индикатор, иначе выключить;

Возвращаемый результат:

- нет (void);

Функции TAU32_LedSet() и TAU32_LedBlink() должны вызываться только для адаптеров, которые были успешно инициализированы. При вызове функции должны быть защищены от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

Текущее состояние светодиодного индикатора отражается в статусе адаптера, см. «TAU32_STATUS» и «TAU32_UserContext{}».

TAU32_SetSyncMode()

```
BOOLEAN TAU32_SetSyncMode(
    TAU32_Controller *pControllerObject,
    unsigned Mode);
```

Функция TAU32_SetSyncMode() устанавливает режим (источник) единой синхронизации адаптера.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- Mode – требуемый режим синхронизации, одна из констант «TAU32_SYNC_MODES»;

Возвращаемый результат:

- ненулевое значение (TRUE), означает, что запрошенный режим синхронизации установлен. Нулевое значение (FALSE), означает, что указан неверный либо неприемлемый для данной модели адаптера режим синхронизации;
- нулевое значение (FALSE), означает, что заданы неверные или недопустимые параметры;

Функция TAU32_SetSyncMode() должна вызываться только для адаптеров, которые были успешно инициализированы. При вызове функция должна быть защищена от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

Если в режимах TAU32_SYNC_RCV_A и TAU32_SYNC_RCV_B источник синхронизации будет потерян, либо выйдет за допустимые пределы, то адаптер автоматически перейдёт на синхронизацию от внутреннего генератора.

В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделам «Устройство адаптера» и «TAU32_SYNC_MODES».

TAU32_SetCrossMatrix()

```
BOOLEAN TAU32_SetCrossMatrix(
    TAU32_Controller *pControllerObject,
    unsigned __int8 *pCrossMatrix,
    unsigned __int32 ReverseMask);
```

Функция TAU32_SetCrossMatrix() устанавливает заданную матрицу кросс-коммутации и маску «переворота» канальных интервалов.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- pCrossMatrix – указатель на образ матрицы кросс-коммутации, массив TAU32_CrossMatrix;

- ReverseMask – битовая маска канальных интервалов, порядок битов в которых необходимо перевернуть;

Возвращаемый результат:

- ненулевое значение (TRUE), означает, что заданные параметры корректны и вступят в силу на ближайшей границе кадра E1 (по источнику синхронизации);
- нулевое значение (FALSE), означает, что заданы неверные или недопустимые параметры;

Функция TAU32_SetCrossMatrix() должна вызываться только для адаптеров, которые были успешно инициализированы. При вызове функция должна быть защищена от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

В процессе работы функция TAU32_SetCrossMatrix() может неоднократно вызывать различные функции обратных вызовов. В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделам «Устройство адаптера» и «Порядок битов в канальных интервалах».

TAU32_SetCrossMatrixCas()

```
BOOLEAN TAU32_SetCrossMatrixCas(
    TAU32_Controller *pControllerObject,
    unsigned __int8 *pCasCrossMatrix);
```

Функция TAU32_SetCrossMatrixCas() устанавливает заданную матрицу кросс-коммутации CAS сигнализации.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- pCasCrossMatrix – указатель на образ матрицы кросс-коммутации, массив TAU32_CrossMatrix. Если этот параметр не равен NULL, то блок обслуживания CAS включается, иначе блок обслуживания CAS отключается;

Возвращаемый результат:

- ненулевое значение (TRUE), означает, что заданные параметры корректны и вступят в силу на ближайшей границе кадра E1 (по источнику синхронизации);
- нулевое значение (FALSE), означает, что заданы неверные или недопустимые параметры;

Функция TAU32_SetCrossMatrixCase() должна вызываться только для адаптеров, которые были успешно инициализированы. При вызове функция должна быть защищена от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

В процессе работы функция TAU32_SetCrossMatrixCas() может неоднократно вызывать различные функции обратных вызовов. В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделам «Устройство адаптера» и «Блок обслуживания сигнализации CAS».

TAU32_SetIdleCodes()

```
BOOLEAN TAU32_SetIdleCodes(
    TAU32_Controller *pControllerObject,
    unsigned __int8 *pIdleCodes);
```

Функция TAU32_SetIdleCodes() устанавливает байтовые значения, передаваемые в неиспользуемых канальных интервалах («idle code»).

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- pIdleCode – указатель на массив байтовых значение, которые будет использоваться в качестве «idle code». Массив должен располагаться в резидентной (NOPAGED) области памяти.
  Каждый элемент массива соответствует одному канальному интервалу, элементы 0-31 соответствуют первому интерфейсу E1, а 32-63 второму (для модели Tau32-Lite используются только первые 32 элемента). Значение каждого элемента массива должно находиться в диапазоне от 0x0 до 0xF включительно и будет использоваться как «idle code» для соответствующего канального интервала. Если элемент массива равен 0xFF, то «idle code» для соответствующего канального интервала не измениться;

Возвращаемый результат:

- ненулевое значение (TRUE), означает, что заданные параметры корректны и функция произвела обновление конфигурации интерфейса E1;
- нулевое значение (FALSE), означает, что заданы неверные или недопустимые параметры;

Функция TAU32_SetIdleCodes() должна вызываться только для адаптеров, которые были успешно инициализированы. При вызове функция должна быть защищена от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

Установленное значение «кода бездействия» будет передаваться в неиспользуемых канальных интервалах в соответствии с «network bit order» (старший бит передаётся первым) вне зависимости от установок «переворота» порядка бит в настройках кросс-коннектора.

В процессе работы функция TAU32_SetIdleCodes() может неоднократно вызывать различные функции обратных вызовов. В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода. За

дополнительной информацией обращайтесь к разделам «Устройство адаптера» и «Неявные действия обслуживания E1».

TAU32_UpdateIdleCodes()

```
BOOLEAN TAU32_UpdateIdleCodes(
    TAU32_Controller *pControllerObject,
    int Interface,
    unsigned __int32 TimeslotMask,
    unsigned __int8 IdleCode);
```

Функция TAU32_UpdateIdleCodes() обновляет (устанавливает) байтовые значения, передаваемые в неиспользуемых канальных интервалах («idle code»).

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- Interface – определяет интерфейс E1, одна из констант «TAU32_INTERFACES»;
- TimeslotMask – битовая маска канальных интервалов, для которых задается новый «код бездействия» («Idle Code»);
- IdleCode – байтовое значение, которое будет использоваться в качестве «idle code»;

Возвращаемый результат:

- ненулевое значение (TRUE), означает, что заданные параметры корректны и функция произвела обновление конфигурации интерфейса E1;
- нулевое значение (FALSE), означает, что заданы неверные или недопустимые параметры;

Функция TAU32_UpdateIdleCodes() должна вызываться только для адаптеров, которые были успешно инициализированы. При вызове функция должна быть защищена от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

Установленное значение «кода бездействия» будет передаваться в неиспользуемых канальных интервалах в соответствии с «network bit order» (старший бит передаётся первым) вне зависимости от установок «переворота» порядка бит в настройках кросс-коннектора.

В процессе работы функция TAU32_UpdateIdleCodes() может неоднократно вызывать различные функции обратных вызовов. В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделам «Устройство адаптера» и «Неявные действия обслуживания E1».

TAU32_SetSaCross()

```
BOOLEAN TAU32_SetSaCross(
    TAU32_Controller *pControllerObject,
    TAU32_SaCross SaCross);
```

Функция TAU32_SetSaCross() производит установку параметров коммутатора Sa-битов.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- SaCross – подготовленная структура TAU32_SaCross, с новыми настройками коммутатора Sa-битов;

Возвращаемый результат:

- ненулевое значение (TRUE), означает, что заданные параметры корректны и функция произвела обновление конфигурации коммутатора Sa-битов;
- нулевое значение (FALSE), означает, что заданы неверные или недопустимые параметры;

Функция TAU32_SetSaCross() должна вызываться только для адаптеров, которые были успешно инициализированы. При вызове функция должна быть защищена от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

В процессе работы функция TAU32_SetSaCross() может неоднократно вызывать различные функции обратных вызовов. В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделам «Коммутатор Sa-бит» и «TAU32_SaCross{}».

TAU32_FifoPutCasAppend(), TAU32_FifoPutCasAhead()

```
int TAU32_FifoPutCasAppend(
    TAU32_Controller *pControllerObject,
    int Interface,
    unsigned __int8 *pBuffer,
    unsigned Length);

int TAU32_FifoPutCasAhead(
    TAU32_Controller *pControllerObject,
    int Interface,
    unsigned __int8 *pBuffer,
    unsigned Length);
```

Обе функции TAU32_FifoPutCasAppend() и TAU32_FifoPutCasAhead() производят пополнение FIFO-буфера используемого для формирования сигнализации CAS при включении режима низкоуровневого обмена.

Функция TAU32_FifoPutCasAppend() пополняет FIFO, дописывая новую порцию данных в его конец («Append»), после текущего содержания.

Функция TAU32_FifoPutCasAhead() пополняет FIFO, вставляя новую порцию данных в его начало («Ahead»), перед текущим содержанием.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- Interface – определяет интерфейс E1, одна из констант «TAU32_INTERFACES»;
- pBuffer – указатель на начало буфера с данными;
- Length – размер добавляемой порции данных в байтах;

Возвращаемый результат:

- значение «-2», если заданы некорректные параметры, при этом FIFO не пополняется;
- значение «-1», если в FIFO недостаточно места для размещения порции данных, при этом FIFO не пополняется;
- другое значение «≥ 0», равно размеру свободного места в FIFO после его пополнения;

Функции TAU32_FifoPutCasAppend() и TAU32_FifoPutCasAhead() должны вызываться только для адаптеров, которые были успешно инициализированы. При вызове функции должна быть защищены от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

В процессе работы функция не вызывает установленные триггеры FIFO или другие функции обратных вызовов. В версии DDK для ОС Windows, обе функции расположены в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделам «Низкоуровневый обмен» и «Блок обслуживания сигнализации CAS».

TAU32_FifoGetCas()

```
int TAU32_FifoGetCas(
    TAU32_Controller *pControllerObject,
    int Interface,
    unsigned __int8 *pBuffer,
    unsigned Length);
```

Функция TAU32_FifoGetCas() производит выборку принятых данных сигнализации CAS из FIFO-буфера, при включенном режиме низкоуровневого обмена.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- Interface – определяет интерфейс E1, одна из констант «TAU32_INTERFACES»;

- pBuffer – указатель на начало буфера для получения данных;
- Length – размер буфера в байтах;

Возвращаемый результат:

- значение «-2», если заданы некорректные параметры, при этом данные из FIFO не выбираются;
- значение «-1», если в FIFO недостаточно данных для полного заполнения буфера, при этом данные из FIFO не выбираются;
- другое значение «≥ 0», равно размеру оставшихся в FIFO данных, после заполнения буфера;

Функция TAU32_FifoGetCas() должна вызываться только для адаптеров, которые были успешно инициализированы. При вызове функция должна быть защищена от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

В процессе работы функция не вызывает установленные триггеры FIFO или другие функции обратных вызовов. В версии DDK для ОС Windows, функция расположены в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделам «Низкоуровневый обмен» и «Блок обслуживания сигнализации CAS».

TAU32_FifoPutFasAppend(), TAU32_FifoPutFasAhead()

```
int TAU32_FifoPutFasAppend(
    TAU32_Controller *pControllerObject,
    int Interface,
    unsigned __int8 *pBuffer,
    unsigned Length);

int TAU32_FifoPutFasAhead(
    TAU32_Controller *pControllerObject,
    int Interface,
    unsigned __int8 *pBuffer,
    unsigned Length);
```

Обе функции TAU32_FifoPutFasAppend() и TAU32_FifoPutFasAhead() производят пополнение FIFO-буфера используемого для формирования значений Sa-битов при включении режима низкоуровневого обмена.

Функция TAU32_FifoPutFasAppend() пополняет FIFO, дописывая новую порцию данных в его конец («Append»), после текущего содержания.

Функция TAU32_FifoPutFasAhead() пополняет FIFO, вставляя новую порцию данных в его начало («Ahead»), перед текущим содержанием.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;

- Interface – определяет интерфейс E1, одна из констант «TAU32_INTERFACES»;
- pBuffer – указатель на начало буфера с данными;
- Length – размер добавляемой порции данных в байтах;

Возвращаемый результат:

- значение «-2», если заданы некорректные параметры, при этом FIFO не пополняется;
- значение «-1», если в FIFO недостаточно места для размещения порции данных, при этом FIFO не пополняется;
- другое значение «≥ 0», равно размеру свободного места в FIFO после его пополнения;

Функции TAU32_FifoPutFasAppend() и TAU32_FifoPutFasAhead() должны вызываться только для адаптеров, которые были успешно инициализированы. При вызове функции должна быть защищены от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

В процессе работы функция не вызывает установленные триггеры FIFO или другие функции обратных вызовов. В версии DDK для ОС Windows, обе функции расположены в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделу «Низкоуровневый обмен».

TAU32_FifoGetFas()

```
int TAU32_FifoGetFas(
    TAU32_Controller *pControllerObject,
    int Interface,
    unsigned __int8 *pBuffer,
    unsigned Length);
```

Функция TAU32_FifoGetFas() производит выборку принятых значений Sa-битов из FIFO-буфера, при включенном режиме низкоуровневого обмена.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- Interface – определяет интерфейс E1, одна из констант «TAU32_INTERFACES»;
- pBuffer – указатель на начало буфера для получения данных;
- Length – размер буфера в байтах;

Возвращаемый результат:

- значение «-2», если заданы некорректные параметры, при этом данные из FIFO не выбираются;

- значение «-1», если в FIFO недостаточно данных для полного заполнения буфера, при этом данные из FIFO не выбираются;
- другое значение «≥ 0», равно размеру оставшихся в FIFO данных, после заполнения буфера;

Функция TAU32_FifoGetFas() должна вызываться только для адаптеров, которые были успешно инициализированы. При вызове функция должна быть защищена от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

В процессе работы функция не вызывает установленные триггеры FIFO или другие функции обратных вызовов. В версии DDK для ОС Windows, функция расположены в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделу «Низкоуровневый обмен».

TAU32_SetFifoTrigger()

```
BOOLEAN TAU32_SetFifoTrigger(
    TAU32_Controller *pControllerObject,
    int Interface,
    unsigned FifoId,
    unsigned Level,
    TAU32_FifoTrigger Trigger);
```

Функция TAU32_SetFifoTrigger() устанавливает адрес функции обратного вызова для уведомления об изменении уровня FIFO-буфера.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- Interface – определяет интерфейс E1, одна из констант «TAU32_INTERFACES»;
- FifoId – определяет FIFO-буфер, одна из констант «TAU32_FIFO_ID»;
- Level – «сигнальное» значение уровня FIFO в байтах, при прохождении через которое будет уведомляться функция обратного вызова. Сигнальное значение имеет различный смысл для FIFO-буферов используемых для приёма и при передаче (см. ниже);
- Trigger – адрес функции обратного вызова, совместимой с типом TAU32_FifoTrigger. Укажите «NULL», для того чтобы отключить триггер;

Возвращаемый результат:

- ненулевое значение (TRUE), означает, что заданные параметры корректны и функция выполнила необходимые действия;
- нулевое значение (FALSE), означает, что заданы неверные или недопустимые параметры;

Функция TAU32_UpdateIdleCodes() должна вызываться только для адаптеров, которые были успешно инициализированы. При вызове функция должна быть защищена от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

Указанное сигнальное значение уровня FIFO имеет различный смысл для FIFO-буферов, используемых для приёма и передачи. Для FIFO-буферов, используемых для приёма, «сигнальное» значение задает контрольный размер свободного места, и функция обратного вызова будет уведомляться, когда размер свободного места будет уменьшаться и пересекает указанную границу.

Для FIFO-буферов, используемых при передаче, «сигнальное» значение задает контрольное количество данных в FIFO, и функция обратного вызова будет уведомляться, когда заполнение FIFO будет уменьшаться и пересекает указанную границу.

В процессе работы функция не вызывает установленные ранее, новые триггеры FIFO или другие функции обратных вызовов. Установленные триггеры FIFO вызываются только при переходе через «контрольное» значение в сторону уменьшения.

В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода. За дополнительной информацией обращайтесь к разделам «Обратные вызовы».

TAU32_ProbeGeneratorFrequency()

```
unsigned __int64 TAU32_ProbeGeneratorFrequency(
    unsigned __int64 Frequency);
```

Функция TAU32_ProbeGeneratorFrequency() позволяет какой будет генерируемая частота с учетом точности и внутренних ограничений генератора.

Частота представляется в $1/2^{32}$ долях герца, или другими словами в формате фиксированной точке 32,32 бит. Например, частоте 2048000,0 Hz соответствует значение 0x001F400000000000.

Параметры:

- Frequency – 64-битное значение, соответствующее частоте (в формате фиксированной точки), не которую необходимо «попробовать» настроить управляемый генератор;

Возвращаемый результат:

- 64-битное значение, соответствующее ближайшей частоте, на которую может быть настроен генератор с учетом точности и внутренних ограничений;

Функция TAU32_ProbeGeneratorFrequency() не использует какие-либо внутренние структуры DDK и не получает указатель на экземпляр объекта TAU32_Controller. Соответственно может вызываться независимо от процесса инициализации DDK, адаптеров Tau32, процессов взаимодействия с ними.

В процессе работы функция не вызывает установленные триггеры FIFO или другие функции обратных вызовов. В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода.

TAU32_SetGeneratorFrequency()

```
unsigned __int64 TAU32_SetGeneratorFrequency(
    TAU32_Controller *pControllerObject,
    unsigned __int64 Frequency);
```

Функция TAU32_SetGeneratorFrequency() производит настройку внутреннего управляемого генератора. Функции не изменяет текущий режим синхронизации адаптера.

Частота представляется в $1/2^{32}$ долях герца, или другими словами в формате фиксированной точке 32,32 бит. Например, частоте 2048000,0 Hz соответствует значение 0x001F400000000000.

Если затребованная частота находится вне допустимых пределов ±5000 Hz относительно базовой частоты (от $2{,}043\times10^6$ Hz до $2{,}053\times10^6$ Hz), то функция настроит генератор на ближайшее допустимое значение.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- Frequency – 64-битное значение, соответствующее частоте (в формате фиксированной точки), не которую необходимо «попробовать» настроить управляемый генератор. Либо значение «0» при этом генератор будет сброшен и установлен на базовую частоту $2{,}048\times10^6$ Hz;

Возвращаемый результат:

- 64-битное значение, соответствующее ближайшей частоте, на которую может быть настроен генератор с учетом точности и внутренних ограничений;

Функция TAU32_SetGeneratorFrequency() должна вызываться только для адаптеров, которые были успешно инициализированы. При вызове функция должна быть защищена от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

В процессе работы функция не вызывает установленные триггеры FIFO или другие функции обратных вызовов. В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода.

TAU32_ReadTsc()

```
void TAU32_ReadTsc(
    TAU32_Controller *pControllerObject,
    TAU32_tsc *pResult);
```

Функция TAU32_ReadTsc() производит чтение счетчиков таковых импульсов опорного тактового генератора и единого внутреннего источника синхронизации адаптера. Чтение обоих значений происходит синхронно, без приостановки работы счетчиков.

Параметры:

- pControllerObject – указатель на объект TAU32_Controller;
- pResult – указатель на экземпляр структуры TAU32_tsc, где будут сохранены считанные значения счетчиков;

Возвращаемый результат:

- нет (void);

Опорный тактовый генератор вырабатывает частоту $2{,}048\times10^{6}$ Hz ±50 ppm. Каждый такт единой внутренней синхронизации соответствует одному принятому/переданному биту в потоках E1.

Функция TAU32_ReadTsc() должна вызываться только для адаптеров, которые были успешно инициализированы. При вызове функция должна быть защищена от прерывания обработчиком прерываний того же адаптера (см. KeSynchronizeExecution() в Windows DDK).

В процессе работы функция не вызывает установленные триггеры FIFO или другие функции обратных вызовов. В версии DDK для ОС Windows, функция расположена в NOPAGED (резидентной) секции программного кода.

Для дополнительной информации см. описание структуры «TAU32_tsc{}», функции «TAU32_SetSyncMode()», а также раздел «Устройство адаптера».

TAU32_IS_REQUEST_RUNNING(), TAU32_IS_REQUEST_NOT_RUNNING()

```
#define TAU32_IS_REQUEST_RUNNING(pUserRequest) \
    ((pUserRequest)->pInternal != NULL)

#define TAU32_IS_REQUEST_NOT_RUNNING(pUserRequest) \
    ((pUserRequest)->pInternal == NULL)
```

Пара макросов позволяющих быстро проверить, выполняется ли запрос или нет.

Параметры:

- pRequest – указатель на структуру TAU32_UserRequest;

При использовании вы должны понимать, что поле pUserRequest→pInternal контролируется DDK, и может быть изменено обработчиком асинхронно в любой момент времени. Кроме этого, необходимо учитывать, что указанная вами функция обратного вызова pUserRequest→pCallback() будет вызвана сразу после обнуления поля pUserRequest→pInternal.

За дополнительной информацией обращайтесь к разделам «Обработка запросов» и «TAU32_UserRequest{}».

Обратные вызовы

TAU32_RequestCallback()

```
typedef void(*TAU32_RequestCallback)
    (TAU32_UserContext *pContext,
     TAU32_UserRequest *pUserRequest);
```

Тип (прототип) функций обратного вызова используемых DDK для уведомления вашего кода о завершении асинхронного выполнения или отмены запроса.

Параметры (передаваемые из DDK):

- pContext – указатель на структуру TAU32_UserContext соответствующую обслуживаемому адаптеру;
- pUserRequest – указатель на запрос, структуру TAU32_UserRequest;

Возвращаемое значение (из вашего кода):

- нет (void);

Адрес этой функции обратного вызова задаётся в поле pUserRequest→pCallback, при формировании запроса, перед его отправкой на исполнение в DDK.

DDK вызовет, указанную вами функцию, один раз при завершении обработки или отмены запроса. Функция не будет вызвана DDK в следующих случаях:

- Запрос был забракован и не принят к исполнению, функция TAU32_SubmitRequest() вернула значение «0» (FALSE);
- Запрос был отменен, и функция TAU32_CancelRequest() подтвердила его синхронную (немедленную) отмену, вернув ненулевое значение (TRUE);
- До момента завершения выполнения запроса, была произведена остановка DDK и адаптера. При этом, функции TAU32_DestructiveHalt(), передавался параметр CancelRequests равный нулю (FALSE);

Предоставленная вами функция обратного вызова должна быть готова к тому, что DDK может вызвать её как из обработчика прерываний TAU32_HandleInterrupt(), так и из любой другой, вызванной вами функции DDK.

Функция может вызывать любые функции DDK кроме TAU32_DestructiveHalt() и TAU32_Initialize(). Однако при вызове функций DDK из вашей функции обратного вызова, может произойти вызов других, заданных вами, функций обратных вызовов. За дополнительной информацией обращайтесь к разделам «Обработка запросов» и «TAU32_UserRequest{}».

TAU32_NotifyCallback()

```
typedef void(*TAU32_NotifyCallback)
    (TAU32_UserContext *pContext,
     int Item,
     unsigned NotifyBits);
```

Тип (прототип) функций обратного вызова используемых DDK для уведомления вашего кода об изменении статуса адаптера или интерфейсов E1, а также о возникновении ошибок приёма/передачи и управления в логических каналах.

Функция, адрес которой указан в поле pUserContext→pErrorNotifyCallback, будет вызываться DDK при возникновении ошибок приёма/передачи и/или управления в логических каналах.

Функция, адрес которой указан в поле pUserContext→ pStatusNotifyCallback, будет вызываться DDK при изменении статуса адаптера или интерфейсов E1.

Параметры (передаваемые из DDK):

- pContext – указатель на структуру TAU32_UserContext соответствующую обслуживаемому адаптеру;
- Item – определяет элемент на котором произошло событие. В зависимости от того, какая из функций вызвана, это может быть либо номер логического канала, либо номер интерфейса E1. Значение «-1» соответствует адаптеру в целом (см. ниже);
- NotifyBits – в зависимости от того, какая из функций вызвана, это может быть либо код ошибки, либо биты изменения статуса (см. ниже);

Возвращаемое значение (из вашего кода):

- нет (void);

Для функции, адрес которой указан в поле pUserContext→pErrorNotifyCallback, значение параметров имеют следующий смысл:

Параметр Item определяет номер логического канала (0..TAU32_CHANNELS), на котором произошла ошибка. Значение «-1» означает, что событие имеет отношение ко всем логическим каналам. Параметр NotifyBits будет содержать информацию об ошибке в виде комбинации битовых флагов TAU32_ERRORS.

Для функции, адрес которой указан в поле pUserContext→ pStatusNotifyCallback, значение параметров имеют следующий смысл:

Параметр Item определяет номер интерфейса E1 («0» или «1»), на котором произошло событие. При этом параметр NotifyBits будет содержать комбинацию битовых флагов TAU32_INTERFACE_STATUS_BITS, которые изменились в статусе интерфейса E1.

Если параметр Item равен «-1», то это означает, что функция вызвана для уведомления об изменении статуса адаптера. Соответственно параметр NotifyBits будет содержать комбинацию битовых флагов TAU32_STATUS, которые изменились в статусе адаптера. Не все изменения в статусе адаптера вызывают уведомление функции обратного вызова, DDK уведомляет функцию обратного вызова pUserContext→pStatusNotifyCallback() по следующим правилам:

- При любом изменении флага TAU32_FRLOMF;

- При деактивации флагов TAU32_CROSS_PENDING или TAU32_CROSS_WAITING;
- О других изменения уведомление не производится;

Предоставленная вами функция обратного вызова должна быть готова к тому, что DDK может вызвать её как из обработчика прерываний TAU32_HandleInterrupt(), так и из любой другой, вызванной вами функции DDK.

Функция может вызывать любые функции DDK кроме TAU32_DestructiveHalt() и TAU32_Initialize(). Однако при вызове функций DDK из вашей функции обратного вызова, может произойти вызов других, заданных вами, функций обратных вызовов. За дополнительной информацией обращайтесь к разделам «Обработка запросов» и «TAU32_UserContext{}».

TAU32_FifoTrigger()

```
typedef void(*TAU32_FifoTrigger)
    (TAU32_UserContext *pContext,
     int Interface,
     unsigned FifoId,
     unsigned Level);
```

Тип (прототип) функций обратного вызова используемых DDK для уведомления вашего кода о пересечении уровнем FIFO-буфера заданного предела.

Параметры (передаваемые из DDK):

- pContext – указатель на структуру TAU32_UserContext соответствующую обслуживаемому адаптеру;
- Interface – определяет интерфейс E1;
- FifoId – определяет FIFO-буфер, одна из констант TAU32_FIFO_ID;
- Level – текущее значение контролируемого уровня в байтах, имеет различный смысл для FIFO-буферов используемых для приёма и при передаче (см. ниже);

Возвращаемое значение (из вашего кода):

- нет (void);

Передаваемое текущее значение контролируемого уровня имеет различный смысл для FIFO-буферов, используемых для приёма и передачи:

- Для FIFO-буферов, используемых для приёма, в параметре Level передается количество данных накопленных в FIFO;
- Для FIFO-буферов, используемых при передаче, в параметре Level передается размер свободного места в FIFO;

Предоставленная вами функция обратного вызова должна быть готова к тому, что DDK может вызвать её как из обработчика прерываний TAU32_HandleInterrupt(), так и из любой другой, вызванной вами функции DDK.

Функция может вызывать любые функции DDK кроме TAU32_DestructiveHalt() и TAU32_Initialize(). Однако при вызове функций DDK из вашей функции обратного вызова, может произойти вызов других, заданных вами, функций обратных вызовов. За дополнительной информацией обращайтесь к разделу «Низкоуровневый обмен».

Критерии аварийных ситуаций E1

Аварийная ситуация	Критерий входа	Критерий выхода	ITU-T
RCL (carrier lost)	Если используется код HDB3: на входе приёмника нет сигнала в течение приёма 255 битов; Если используется код AMI: последовательно принято 2048 или более нулей;	Если используется код HDB3: на входе приёмника есть сигнал в течение приёма 32 битов; Если используется код AMI: из 255 последних принятых бит 32 не нулевых;	G.775, G.962
RUA1 (unframed all ones)	Среди последних принятых 512 битов менее 3 нулей;	Среди последних принятых 512 битов 3 или более нулей;	O.162 §1.6.1.2
RRA (remote alarm)	Бит 3 нечетного (NAF) фрейма E1 установлен в «1» три раза подряд;	Бит 3 нечетного (NAF) фрейма E1 установлен в «0» три раза подряд;	O.162 §2.1.4
RSA1 (signaling all ones)	Во всех байтах последнего сверхцикла 16ый канальный интервал содержал менее 3 нулей;	Во всех байтах последнего сверхцикла 16ый канальный интервал содержал 3 или более нуля;	G.732 §4.2
RSA0 (signaling all zeros)	Во всех байтах последнего сверхцикла 16ый канальный интервал содержал менее 3 единиц;	Во всех байтах последнего сверхцикла 16ый канальный интервал содержал 3 или более единицы;	G.732 §5.2
RDMA (distant multiframe alarm)	Бит 6 первого байта 16го канального интервала был установлен в «1» для двух или более последовательных сверхциклов;	Бит 6 первого байта 16го канального интервала был установлен в «0» для двух или более последовательных сверхциклов;	O.162 §2.1.5
RFAS (FAS sync loss)	Приняты три последовательных неправильных синхросигнала, или бит 2 в нулевом канальном интервале в нечетных (NAF) фреймах принят с ошибкой три раза подряд;	Синхросигнал обнаружен в фреймах N и N-2, и не обнаружен в фрейме N-1;	G.706 §4.1.1, §4.1.2
RCRC4 (CRC5 sync loss)	Более 914 блоков CRC4 из 1000 последних принято с ошибками;	За последние 8 ms принято два верных сверхцикла CRC4, с интервалом кратным 2 ms;	G.706 §4.2, §4.3.2
RCAS (CAS sync loss)	Два последовательных сверхцикла CAS приняты с ошибками;	Найден правильный синхросигнал CAS и байт 16го канального интервала в предыдущем фрейме E1 не равен нулю;	G.732 §5.2
$RCRC4_{LONG}$	Авария RCRC4 в течение 400 ms или более;	Нет аварии CRC4;	G.706 §4.2
RLOS (loss of sync)	Присутствует хотя бы одна из аварий: RFAS, RCRC4, RCAS	Нет ни одной из аварий: RFAS, RCRC4, RCAS	-

Примеры

ОС Linux (для версии 2.6.x)

Общие структуры и типы

```
#define TAU32_UserRequest_Add void *pTag; // поле pTag будет добавлено
                                          // к структуре TAU32_UserRequest
#include "tau32-ddk.h"                    // подключаем DDK

// адаптер Tau32
typedef struct _tau32_board_t
{
    TAU32_UserContext ddk;          // контекст ddk, на первом месте для
                                       возможности прямого преобразования
                                       указателей
    wait_queue_head_t wait;         // "точка" ожидания завершения
                                       управляющих запросов
    struct pci_dev dev;             // копия структуры
    int running;                    // флаг - запущен адаптер или нет
    unsigned long intr_count;       // счетчик прерываний
    spinlock_t lock;                // spinlock для синхронизации с
                                       обработчиком прерываний
    tau32_chan_t ch[TAU32_CHANNELS];
    int num;                        // номер адаптера в системе
    int irq;                        // линия IRQ выделенная адаптеру
    unsigned long base;             // базовый аппаратный адрес регистров
                                       адаптера
} tau32_board_t;

// логический канал Tau32
typedef struct _tau32_chan_t
{
    struct _tau32_board_t *board;
    int num;                        // номер логического канала на адаптере
    unsigned config;                // конфирурация
    unsigned long ts;               // маска привязанных канальных интервалов
                                    // счетчики:
    unsigned long obytes;           // переданно байтов
    unsigned long opkts;            // переданно пакетов
    unsigned long ibytes;           // принято байтов
    unsigned long ipkts;            // принято пакетов
    unsigned long error_underrun;   // ошибок вида "underrun"
    unsigned long error_overrun;    // ошибок вида "overrun"
    unsigned long error_frame;      // ошибок кадров HDLC
    unsigned long error_crc;        // ошибок контрольной суммы HDLC
} tau32_chan_t;

// буфер-запрос на приём/передачу данных
typedef struct _tau32_buf_t
{
    TAU32_UserRequest request;      // запрос к ddk, на первом месте для
                                       возможности прямого преобразования
                                       указателей
    unsigned char data[IO_SIZE];    // буфер под данные
    char hdlc_gap[4];               // необходимый запас для приёма
                                       HDLC-пакетов
} tau32_buf_t;
```

Инициализация

```
tau32_board_t* tau32_init(void)
/*
    Находит адаптер, запрашивает необходимые ресурсы и подготавливает
    его к работе. Если установленно несколько адаптеров используется первый.
*/
{
    tau32_board_t *b;
    unsigned long dev_hw_addr;
    void *dev_virt_addr;
    struct pci_dev *dev;
    int i;

    // получаем первое совместимое PCI-устройство
    dev = pci_find_device(TAU32_PCI_VENDOR_ID, TAU32_PCI_DEVICE_ID, 0);
    if(! dev)
        return NULL;

    b = kmalloc(sizeof(tau32_board_t), GFP_KERNEL);
    if(!b)
    {
        printk("out of memory %d\n", sizeof(tau32_board_t));
        return NULL;
    }
    memset(b, 0, sizeof(tau32_board_t));

    // инициализируем spin_lock
    spin_lock_init(&b->lock);
    // инициализируем массив логических каналов
    for(i = 0; i < TAU32_CHANNELS; i++)
    {
        tau32_chan_t *c = b->ch + i;
        c->num = i;
        c->board = b;
    }

    // выделяем память для DDK
    b->ddk.pControllerObject = kmalloc(TAU32_ControllerObjectSize,
                                       GFP_KERNEL);
    if(!b->ddk.pControllerObject)
    {
        printk("out of memory %d\n", TAU32_ControllerObjectSize);
        tau32_cleanup(b);
        return NULL;
    }

    // получаем физический адрес регистров адаптера,
    // захватывем диапазон адресов
    dev_hw_addr = pci_resource_start(dev, 0);
    if(check_mem_region(dev_hw_addr, TAU32_PCI_IO_BAR1_SIZE) != 0)
    {
        printk(KERN_ERR "adapter busy at mem 0x%p\n",(void*) dev_hw_addr);
        tau32_cleanup(b);
        return NULL;
    }
    request_mem_region(dev_hw_addr, TAU32_PCI_IO_BAR1_SIZE, "CronyxTau32");

    // отображаем регистры адаптера в виртуальное адресное пространство
    dev_virt_addr = ioremap_nocache(dev_hw_addr, TAU32_PCI_IO_BAR1_SIZE);
```

```
    if(! dev_virt_addr)
    {
        printk("cannot ioremap_nocache pci memory 0x%p\n",(void*)
                                        dev_hw_addr);
        tau32_cleanup(b);
        return NULL;
    }

    // заполняем поля структуры TAU32_UserContext
    b->ddk.PciBar1VirtualAddress = dev_virt_addr;
    b->ddk.ControllerObjectPhysicalAddress =
                                        virt_to_bus(b->ddk.pControllerObject);
    b->ddk.pErrorNotifyCallback = tau32_channel_notify;
    b->ddk.pStatusNotifyCallback = tau32_interface_notify;
    b->irq = dev->irq;
    b->base = dev_hw_addr;
    b->dev = *dev;

    // инициализируем "точку" ожидания
    init_waitqueue_head(&b->wait);

    // разрешаем и настраиваем доступ к устройству с PCI-шины,
    // b->dev.irq будет соответствовать логической линии irq
    if(pci_enable_device(&b->dev))
    {
        printk("pci_enable_device error\n");
        tau32_cleanup(b);
        return NULL;
    }
    pci_set_master(&b->dev);

    // инициализируем DDK и адаптер
    if(!TAU32_Initialize(&b->ddk, 0))
    {
        printk("init adapter error 0x%08x, dead bits 0x%08lx\n",
            b->ddk.InitErrors, b->ddk.DeadBits);
        tau32_cleanup(b);
        return NULL;
    }

    // подключаем обработчик прерываний к линии IRQ
    if(request_irq(b->dev.irq, tau32_isr, SA_SHIRQ, "CronyxTau32", b) != 0)
    {
        printk("can't get irq %d/%d\n", b->irq, b->dev.irq);
        tau32_cleanup(b);
        return NULL;
    }
    b->running = 1;

    // разрешаем генерацию прерываний,
    // синхронизация с обработчиком прерыванию еще не нужна,
    // так как генерация прерываний еще не разрешена
    TAU32_EnableInterrupts(b->ddk.pControllerObject);
    return b;
}
```

Остановка

```
void tau32_cleanup(tau32_board_t *b)
```

```
/*
    Останавливает адаптер и высвобождает занятые ресурсы
*/
{
    if(!b)
        return;
    if(b->running)
    {
        // адаптер запущен, необходимо остановить его
        context_t l;
        spin_lock_irqsave(&b->lock, l); // синхронизация с обработчиком
                                        прерываний

        // запрещаем генерацию аппаратных прерываний
        TAU32_DisableInterrupts(b->ddk.pControllerObject);
        spin_unlock_irqrestore(&tau32_lock, l);

        // отключаем обработчик прерываний
        if(b->irq)
            free_irq(b->irq, b);

        // останавливаем адаптер и DDK,
        // синхронизация с обработчиком прерыванию уже не нужна,
        // так как генерация прерываний уже запрещена
        TAU32_DestructiveHalt(b->ddk.pControllerObject, 1);
        b->running = 0;
    }

    // снимаем отображение регистров адаптера в память
    if(b->ddk.PciBar1VirtualAddress)
        iounmap(b->ddk.PciBar1VirtualAddress);

    // разблокируем диапазон адресов
    if(b->base)
        release_mem_region(b->base, TAU32_PCI_IO_BAR1_SIZE);

    // чистим и освобождаем оперативную память
    if(b->ddk.pControllerObject)
    {
        memset(b->ddk.pControllerObject, 0xFF, TAU32_ControllerObjectSize);
        kfree(b->ddk.pControllerObject);
    }
    memset(b, 0xFF, sizeof(tau32_board_t));
    kfree(b);
}
```

Обработка прерываний

```
irqreturn_t tau32_isr(int irq, void *dev, struct pt_regs *regs)
/*
    Обработчик прерываний
*/
{
    tau32_board_t *b = (tau32_board_t*) dev;

    // синхронизация с вызовом других функций
    spin_lock(&b->lock);

    if(TAU32_HandleInterrupt(b->ddk.pControllerObject))
```

```
    {
        // было прерывание от адаптера, увеличиваем счетчик
        b->intr_count++;
        spin_unlock(&b->lock);
        return IRQ_HANDLED;
    }

    spin_unlock(&b->lock);
    return IRQ_NONE;
}
```

Обратные вызовы по изменению статуса и ошибкам

```
void TAU32_CALLBACK_TYPE tau32_interface_notify(TAU32_UserContext *v, int
                                        Interface, unsigned Delta)
/*
    Функция обратного вызова по изменению статуса интерфейсов E1 или адаптера
    TAU32_NotifyCallback()
*/
{
    if(Interface >= 0)
    {
        // статус интерфейса E1
        printk("interface %d status bits 0x%08x delta 0x%08x\n",
            Interface, pContext->InterfacesInfo[Interface].Status, Delta);
    }
    else
    {
        // статус адаптера
    }
}

void TAU32_CALLBACK_TYPE tau32_channel_notify(TAU32_UserContext *v, int
                                        Channel, unsigned Bits)
/*
    Функция обратного вызова для уведомления об ошибках
    логических каналов или адаптера
    TAU32_NotifyCallback()
*/
{
    if(Channel >= 0)
    {
        // ошибка логического канала
        tau32_board_t *b = (tau32_board_t *) v;
        tau32_chan_t *c = &b->ch[Channel];
        printk("Channel %d/%d notify error bits 0x%08x",
            b->num, c->num, Bits);

        if(Bits & TAU32_ERROR_TX_UNDERFLOW)
            c->error_underrun++;
        if(Bits & TAU32_ERROR_RX_OVERFLOW)
            c->error_overrun++;
        if(Bits & (TAU32_ERROR_TX_PROTOCOL | TAU32_ERROR_RX_ABORT |
                                        TAU32_ERROR_RX_SHORT |
                                        TAU32_ERROR_RX_FRAME |
                                        TAU32_ERROR_RX_LONG |
                                        TAU32_ERROR_RX_SYNC))
            c->error_frame++;
        if(Bits & TAU32_ERROR_RX_CRC)
```

```
            c->error_crc++;

        // TODO: Ваш код
    }
    else
    {
        // ошибка адаптера
    }
}
```

Управляющие запросы

```
void TAU32_CALLBACK_TYPE tau32_request_callback(TAU32_UserContext *v,
                                                TAU32_UserRequest *r)
/*
    Функция обратного вызова для управляющих запросов.
    DDK будет вызывать эту функцию по завершении асинхронной обработки.
    TAU32_RequestCallback()
*/
{
    // в самом начале мы добавили поле pTag к структуре TAU32_UserRequest
    // именно для того, чтобы использовать его здесь
    if(r->pTag)
        wake_up_interruptible((wait_queue_head_t*) r->pTag);
}

int tau32_submit_request_and_wait(tau32_board_t *b, TAU32_UserRequest *r)
/*
    Посылалает к DDK запрос и ждет его завершения,
    при необходимости производит отмену запроса
*/
{
    context_t l;
    r->pTag = &b->wait; // задаем адрес объекта wait_queue_head
    r->pCallback = tau32_request_callback; // определяем функцию обратного
                                           вызова
    spin_lock_irqsave(&b->lock, l); // синхронизация с обработчиком
                                       прерываний
    if(!TAU32_SubmitRequest(b->ddk.pControllerObject, r))
    {
        spin_unlock_irqrestore(&b->lock, l);
        printk("submit request and wait, error\n");
        return -1;
    }
    spin_unlock_irqrestore(&b->lock, l);
    // запрос принят к обработке, ждем завершения выполнения
    // по завершению DDK вызовет tau32_request_callback()
    // и wait_event_interruptible() проснется
    if(wait_event_interruptible(b->wait, TAU32_IS_REQUEST_NOT_RUNNING(r)))
    {
        // прерывания действия (нажато ^C), необходимо отметить запрос
        spin_lock_irqsave(&b->lock, l); // синхронизация с обработчиком
                                           прерываний
        if(TAU32_IS_REQUEST_RUNNING(r)
        && ! TAU32_CancelRequest(b->ddk.pControllerObject, r, 1))
        {
            // запрос еще выполняется и не может быть отменен немедленно,
            // ждем подтверждения отмены
            spin_unlock_irqrestore(&b->lock, l);
```

```
            wait_event(b->wait, TAU32_IS_REQUEST_NOT_RUNNING(r));
        }
        else
        {
            // запрос уже не выполняется или успешно отменен
            spin_unlock_irqrestore(&b->lock, l);
        }

        printk("submit request and wait, cancelled\n");
        return TAU32_ERROR_CANCELLED;
    }
    return r->ErrorCode;
}
```

Передача данных

```
void tau32_submit_tx(tau32_chan_t *c, tau32_buf_t *t)
/*
    Направляем запрос на передачу данных
*/
{
    context_t l;

    t->request.Command = TAU32_Tx_Data;   // команда
    t->request.Io.ChannelNumber = c->num; // номер логического канала
    t->request.pCallback = tau32_callback_tx;   // функция обратного вызова
    t->request.Io.Tx.DataLength = sizeof(t->data);    // размер

    // физический адрес чтения передаваемых данных
    t->request.Io.Tx.PhysicalDataAddress = virt_to_bus(&t->data);

    spin_lock_irqsave(&c->board->lock, l); // синхронизация с обработчиком
                                           прерываний
    if(!TAU32_SubmitRequest(c->board->ddk.pControllerObject, r))
        printk("error submit-tx\n");
    spin_unlock_irqrestore(&c->board->lock, l);
}

void TAU32_CALLBACK_TYPE tau32_callback_tx(TAU32_UserContext *v,
                                           TAU32_UserRequest *r)
/*
    Функция обратного вызова для запросов на передачу данных.
    DDK будет вызывать эту функцию по завершении асинхронной обработки.
    TAU32_RequestCallback()
*/
{
    tau32_board_t *b = (tau32_board_t *) v;
    tau32_buf_t *t = (tau32_buf_t *) r;
    tau32_chan_t *c = &b->ch[t->request.Io.ChannelNumber];

    c->obytes += t->request.Io.Tx.Transmitted;
    c->opkts++;

    if(t->request.ErrorCode)
    {
        printk("Channel %d/%d tx-packet #%ld tx-error 0x%0lX",
            c->board->num, c->num, c->opkts, t->request.ErrorCode);
    }
    else if(t->request.Io.Tx.Transmitted != t->request.Io.Tx.DataLength)
```

```
    {
        printk("Channel %d/%d tx-packet #%ld tx-incomplete length %d should
                                        be %d",
            c->board->num, c->num, c->opkts,
            t->request.Io.Tx.Transmitted, t->request.Io.Tx.DataLength);
    }
    // TODO: Ваш код
}
```

Приём данных

```
void tau32_submit_rx(tau32_chan_t *c, tau32_buf_t *t)
/*
    Направляем запрос на приём данных
*/
{
    context_t l;

    t->request.Command = TAU32_Rx_Data;         // команда
    t->request.Io.ChannelNumber = c->num;       // номер логического канала
    t->request.pCallback = tau32_callback_rx;   // функция обратного вызова;
    t->request.Io.Rx.BufferLength = sizeof(t->data); // размер
    if((c->config & TAU32_channel_mode_mask) == TAU32_HDLC)
        // в режиме HDLC размер буфера должен быть на 4-ре байта
        // больше максимального размера принимаемого пакета
        t->request.Io.Rx.BufferLength += sizeof(t->hdlc_gap);
    // физический адрес для записи принимаемых данных
    t->request.Io.Rx.PhysicalDataAddress = virt_to_bus(&t->data);

    spin_lock_irqsave(&c->board->lock, l); // синхронизация с обработчиком
                                        прерываний
    if(!TAU32_SubmitRequest(c->board->ddk.pControllerObject, &f->request))
        printk("error submit-rx\n");

    spin_unlock_irqrestore(&c->board->lock, l);
}

void TAU32_CALLBACK_TYPE tau32_callback_rx(TAU32_UserContext *v,
                                        TAU32_UserRequest *r)
/*
    Функция обратного вызова для запросов на приём данных.
    DDK будет вызывать эту функцию по завершении асинхронной обработки.
    TAU32_RequestCallback()
*/
{
    tau32_board_t *b = (tau32_board_t *) v;
    tau32_buf_t *t = (tau32_buf_t *) r;
    tau32_chan_t *c = &b->ch[t->request.Io.ChannelNumber];

    c->ipkts++;
    c->ibytes += t->request.Io.Rx.Received;

    if(t->request.ErrorCode)
    {
        printk("Channel %d/%d rx-packet #%ld rx-error 0x%0lX",
            c->board->num, c->num, c->ipkts, t->request.ErrorCode);
    }
    else
        if(!t->request.Io.Rx.FrameEnd
```

```
        && (c->config & TAU32_channel_mode_mask) == TAU32_HDLC)
        {
            printk("Channel %d/%d rx-packet #%ld is not a frame",
                c->board->num, c->num, c->ipkts);
        }
    // TODO: Ваш код
}
```

Конфигурирование

```
int tau32_configure(tau32_board_t *b)
/*
    Задаем некоторую работоспособную конфигурацию
*/
{
    context_t l;
    TAU32_UserRequest R;
    int i;
    TAU32_CrossMatrix CrossMatrix;

    // устанавливаем синхронизацию от интерфейса E1/0
    spin_lock_irqsave(&b->lock, l); // синхронизация с обработчиком
                                    прерываний
    if(!TAU32_SetSyncMode(b->ddk.pControllerObject, TAU32_SYNC_RCV_A))
    {
        spin_unlock_irqrestore(&b->lock, l);
        return -EIO;
    }
    spin_unlock_irqrestore(&b->lock, l);

    // задаем режим работы интерфейсов E1
    R.Command = TAU32_Configure_E1;
    R.Io.InterfaceConfig.Interface = TAU32_E1_ALL;
    R.Io.InterfaceConfig.Config = TAU32_LineNormal | TAU32_framed_cas_set |
                                    TAU32_crc4_mf;
    R.Io.InterfaceConfig.UnframedTsMask = uf_ts_mask;
    if(tau32_submit_request_and_wait(b, &R))
        return -EIO;

    // определяем кросс-коммутацию канальных интервалов
    // обмен по всем каналым интервалам между
    // приёмопередатчиком и интерфейсом E1/0
    for(i = 0; i < TAU32_CROSS_WIDTH; i++)
    {
        if(i < 32)
            CrossMatrix[i] = i + 32; // E1/0 => приёмопередатчик
        else if(i < 64)
            CrossMatrix[i] = i - 32; // приёмопередатчик => E1/0
        else
            CrossMatrix[i] = TAU32_CROSS_OFF; // выключить обмен
    }

    // устанавливаем сформированную матрицу кросс-коммутации
    // и задаем для всех канальных интервалов кроме 16-го
    // HDLC-порядок бит (младший передается первым)
    spin_lock_irqsave(&b->lock, l); // синхронизация с обработчиком
                                    прерываний
    if(!TAU32_SetCrossMatrix(b->ddk.pControllerObject, CrossMatrix,
                                    0x00010001ul))
```

```
    {
        spin_unlock_irqrestore(&b->lock, l);
        return -EIO;
    }
    spin_unlock_irqrestore(&b->lock, l);

    // задаем привязку канальных интервалов к 31-му логическому каналу
    R.Command = TAU32_Timeslots_Channel;
    R.Io.ChannelNumber = 31;
    R.Io.ChannelConfig.AssignedTsMask = 0xFFFEFFFE // все канальные интервалы
                                        кроме 0 и 16
    if(tau32_submit_request_and_wait(b, &R))
        return -EIO;

    // задаем конфигурацию 31-го логического канала и запускаем его
    R.Command = TAU32_Configure_Channel | TAU32_Rx_Start | TAU32_Tx_Start;
    R.Io.ChannelNumber = 31;
    R.Io.ChannelConfig.Config = TAU32_HDLC | TAU32_fr_rx_splitcheck |
                                        TAU32_fr_rx_fitcheck |
                                        TAU32_fr_tx_auto;
    if(tau32_submit_request_and_wait(b, &R))
        return -EIO;
    b->ch[31].config = R.Io.ChannelConfig.Config;

    // задаем привязку канальных интервалов к 0-му логическому каналу
    R.Command = TAU32_Timeslots_Channel;
    R.Io.ChannelNumber = 0;
    R.TimeslotsAssignment.Mask = 0x00010000ul; // только 16-й канальный
                                        интервал
    if(tau32_submit_request_and_wait(b, &R))
        return -EIO;

    // задаем конфигурацию 0-го логического канала и запускаем его
    R.Command = TAU32_Configure_Channel | TAU32_Rx_Start | TAU32_Tx_Start;
    R.Io.ChannelNumber = 0;
    R.Io.ChannelConfig.Config = TAU32_TMA;
    if(tau32_submit_request_and_wait(b, &R))
        return -EIO;
    b->ch[0].config = R.Io.ChannelConfig.Config;

    // актуализируем заданные изменения в конфигурации
    R.Command = TAU32_Configure_Commit;
    if(tau32_submit_request_and_wait(b, &R))
        return -EIO;

    return 0;
}
```

История изменений и исправления ошибок

Версия 1.1, Январь 2005

Добавлено описание и поддержка модели адаптера Tau32-PCI/Lite.

Добавлены функции и связанные с ними константы для взаимодействия с управляемым генератором частоты синхронизации «TAU32_ProbeGeneratorFrequency()» и «TAU32_SetGeneratorFrequency()», а также функция чтения внутренних счетчиков времени «TAU32_ReadTsc()».

Описана ошибка дизайна контроллера «Infineon MUNICH32X» приводящая к вероятностному «сдвигу битов» при приёме в «прозрачном режиме», проблема помечена как «изучаемая».

Версия 1.2, Апрель 2005

Добавлена функция «TAU32_BeforeReset()» для обхода аппаратной ошибки контроллера «Infineon MUNICH32X», которая могла приводить к генерации неожидаемого сигнала аппаратного прерывания при сбросе приёмопередатчика через документированные PCI-регистры.

Реализован обход, обозначенной ранее как «изучаемой», проблемы вероятностного «сдвига битов» (несовпадений границ канальных интервалов и байтов) при приёме в «прозрачном режиме».

Исправлена ошибка внутреннего firmware и ошибка DDK, вследствие которых внутренне состояние матрицы кросс-коммутации не полностью соответствовало записанным данным.

Версия 1.3, Июль 2005

Исправлена ошибка, вследствие которой была маловероятная возможность «застревания» завершенного запроса на приём данных (приятого пакета) до приёма следующей порции данных. Ситуация возникала при следующем стечении обстоятельств:

- в момент завершения заполнения буфера и перехода к следующему возникала активность на шине PCI и по этой причине возникали задержки в обращении к основной памяти компьютера со стороны адаптера Tau32;
- время выполнения центральным процессором кода обработчика прерывания в пути приёма данных перекрывалось с вынужденной паузой в доступе к PCI шине со стороны адаптера;
- вызывающий код всегда обеспечивал непустую очередь запросов на приём данных;

Такая ситуация могла возникать независимо для каждого логического канала и сохранялась либо до перезапуска этого канала, либо до опустошения очереди за-

просов на приём данных. Во входящей очереди, таким образом, могло задерживаться не более одного запроса приёма данных для каждого логического канала.

Дополнены режимы работы интерфейсов E1, добавлены константы TAU32_crc4_mf_rx_only и TAU32_crc4_mf_tx_only. Переопределены многие другие константы связанные с указанием режима работы интерфейсов E1.

Бинарной совместимости с предыдущими версиями DDK **нет**, совместимость на уровне исходных текстов **есть**.

Версия 1.4, Февраль 2006

Устранена возможность маловероятной ситуации «гонок» (race conditions) между DMA контроллера Infineon MUNICH32X и центральным процессором компьютера. В критические участки кода, при обмене с разделяемой памятью, добавлен принудительный сброс отложенной очереди записи со стороны процессора, а также обновление данных с помощью явных interlocked-инструкций. Данная проблема воспроизводилась обычно не регулярно и только на некоторых чипсетах (материнских платах).

Дополнены режимы работы интерфейсов E1, добавлены константы TAU32_framed_cas_pass, TAU32_sa_bypass и TAU32_si_bypass. Переопределены многие другие константы связанные с указанием режима работы интерфейсов E1.

Бинарной совместимости с предыдущими версиями DDK **нет**, совместимость на уровне исходных текстов **есть**.

Версия 1.5, Май 2006

Устранена ошибка излишнего замещения данных, которые пробрасываются в канальных интервалах между интерфейсами E1 средствами кросс-коннектора.

Такое замещение (кодом «все единицы») производиться, только когда источник данных находится в аварийной ситуации. Однако из-за ошибки замещение не отключалось в случае, когда оба интерфейса E1 переходили в/из состояния аварии и очередность входа в аварийное состояние не совпадала с очередностью выхода (для интерфейсов E1).

Бинарной совместимости с предыдущими версиями DDK **нет**, совместимость на уровне исходных текстов **есть**.

Версия 1.6, Июль 2006

Для реализации функциональности DACS в модели Tau-PCI/32 реализован независимый кросс-коннектор CAS. Добавлена функция TAU32_SetCrossCas(), удалена функция TAU32_SetCasIo(). Переименованы константы связанные со статусом адаптера.

Бинарной совместимости с предыдущими версиями DDK **нет**, совместимости на уровне исходных текстов **нет**.

