FPGA SYSTEMS

Разработка цифровой аппаратуры нетрадиционным методом: Контроллер USB 1.0 на SpinalHDL

29/11/2025

Руслан Залата

- Директор ООО «Фабмикро», г. Тюмень
- Энтузиаст нетрадиционных технических решений
- rz@fabmicro.ru

// План доклада

FPGA SYSTEMS
2025

I. Вводна часть

1. Плата «Карно» для синтезируемой ЭВМ (RISC-V)
2. Зачем потребовался USB 1.0 контроллер на SpinalHDL

II. Краткое описание стандарта USB 1.0

3. USB 1.0 на физическом уровне
4. USB 1.0 на уровне представления данных
5. Виды транзакций USB 1.0
6. Древовидная структура конфигурационных данных
7. Процедура инициализация USB устройства (Enumeration)
8. USB Human Interface Device Class

III. Описание КА USB 1.0 хост-контроллера

9. Требования к USB хост-контроллеру
10. Структура USB хост-контроллера
11. Основной КА USBMain
12. КА USBSendSE0 для выполнения сброса и «KeepAlive»
13. КА USBSendShortToken для передачи короткого токена
14. КА USBSendLongToken для передачи длинного токена
15. КА USBSendData для передачи пакета данных DATAx
16. КА USBReceiver для приема произвольного пакета данных
17. Реализация КА хост-контроллера на SpinalHDL
18. Реализация основного КА и логики управления
19. Интеграция хост-контроллера в СнК
20. Сборка проекта (синтез битстрима)

VI. Разработка драйвера и прикладного ПО

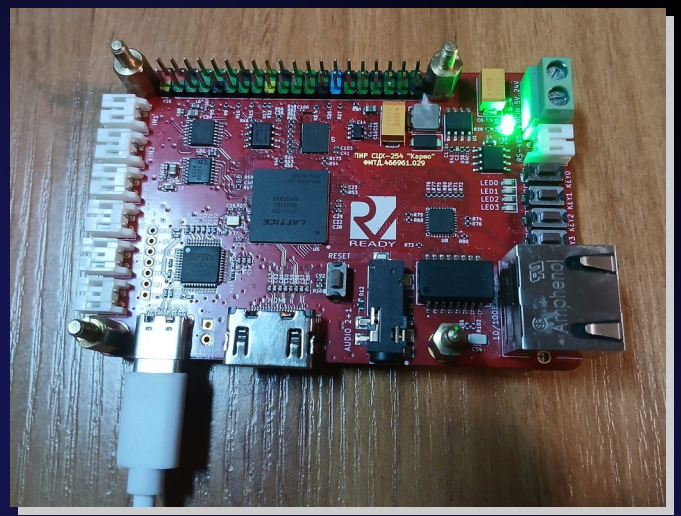
... to be continued.

// 1. Плата «Карно» для синтезируемой ЭВМ (RISC-V)

FPGA SYSTEMS
2025

В 2023 году в процессе работы над добавлением поддержки опенсорсного тулчейна (Yosys) в проект **basics_graphics_music** родилась идея своей отладочной платы на базе ПЛИС **Lattice ECP5**:

1. Встроенный программатор с USB интерфейсом на базе FT2232 (порт A: JTAG, порт B: UART).
2. Интерфейса FastEthernet.
3. Видео с нтерфейсом HDMI.
4. Микросхема статической памяти (мин 512КБ).
5. ЦАП, АЦП, RS-485.
6. Питание от +12...24 В, форм-фактор RPi 3 с совместимым разъемом GPIO.



Цель - разработать синтезируемую СнК на базе опенсорсного ядра RISC-V для промышленного применения.

- ПЛИС **Lattice iCE40** и **ECP5** одни из первых были зареверсены, на них вырос Yosys.

- Разработка платы «Карно», с изготовлением трех ревизий, заняла чуть больше месяца. Через два месяца была изготовлена установочная партия, все платы разошлась по рукам.

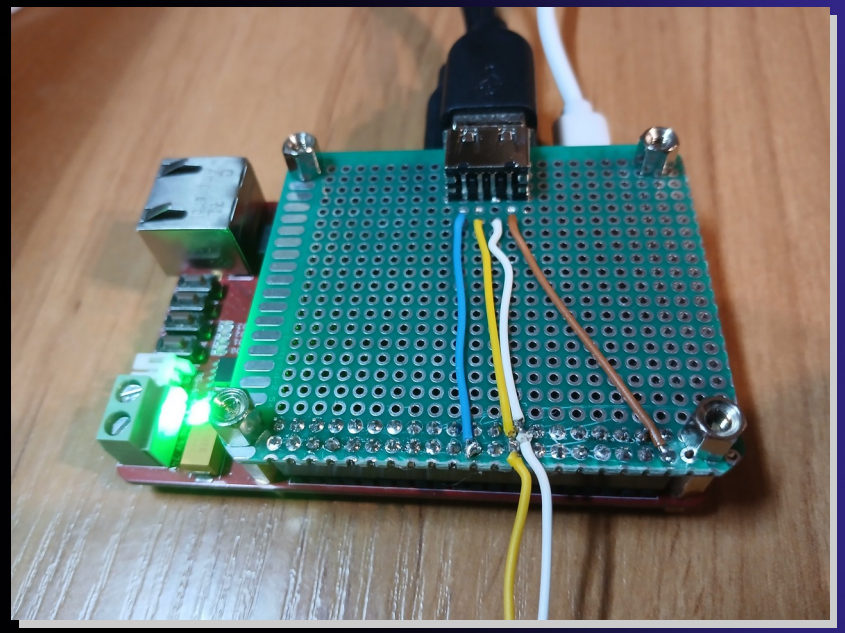
- Дмитрий Петренко придумал название «ПИР СЦХ-254 «**Карно**» (что-то там про симуляцию ASIC), сокращенно «Карно» (англ. «Karnix»).

// 2a. Зачем потребовался USB 1.0 контроллер на SpinalHDL ?

FPGA SYSTEMS
2025

Разработка синтезируемой СнК «Карно» (KarnixSoC) ведется на **SpinalHDL**:

- **VexRiscV** (RV32IMFAC + MMU) в качестве ядра @ 60+ МГц.
- Подключены MAC (FastEthernet), ЦАП и АЦП (SPI).
- Разработан CGA-подобный видео-контроллера с выводом на HDMI.
- Разработан контроллер QSPI с поддержкой XiP — 15 MiB свободной NOR Flash под программы.
- HAL на Си.
- Загрузчик Bootloader (ROM).
- Программа «Монитор» для загрузки и отладки ПО.
- В качестве примера - игра «TetRISC-V».



Для организации интерфейса с пользователем (HMI) потребовалась клавиатура и «мышь». А для игр - «джойстик».

От PS/2 отказался, так как не современно и не спортивно.
Захотелось USB HID.

Приделал к «Карно» «шляпу» с разъемом **USB type A** и приступил к изучению вопроса.

// 2c. Зачем потребовался USB 1.0 контроллер на SpinalHDL ?

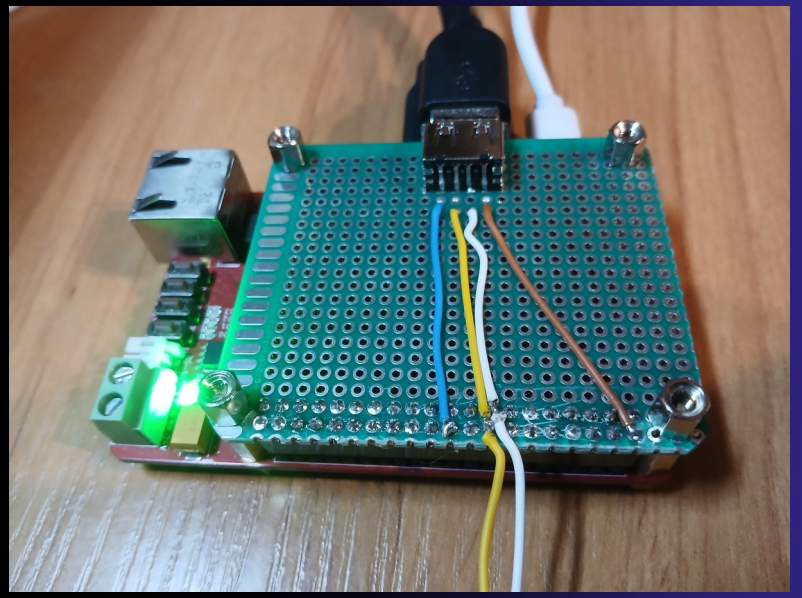
FPGA SYSTEMS 2025

Что есть из доступного:

1. В репозитории SpinalHDL имеется своя реализация USB 1.1 (OHCI совместимая):
- Требует наличие DMA, который у меня еще не реализован.
- Требователен к ресурсам ПЛИС (не влезает в Lattice ECP5 25F).
- Используются аппаратные блоки (SERDES) — нежелательная и ненужная зависимость.

2. На Github имеется ряд тяжелых реализаций на Verilog с использованием проприетарных IP-блоков и привязанных к проприетарным тулам — «не айс!»

3. На «Hackaday» прочел про проект японского мастера «**nand2mario**»:
- Поддерживает HID клавиатуру, «геймпад» и «мышь».
- 600 строк отборной криптографии на Verilog-е без пояснений.
- 400 строк на своём ассемблере для своей машины. **=8-[ ]**
- Не проверяет CRC5 и CRC16.
- Не делает коррекцию частоты по преамбуле (SYNC).
- Как результат - крайне нестабилен, часто сыпет «мусор» вместо кодов клавиш.



Принял решение писать своё. Чем я хуже японца ?

// 2d. Зачем потребовался USB 1.0 контроллер на SpinalHDL ?

FPGA SYSTEMS
2025

План — написать «MVP» реализацию USB 1.0 хост-контроллера на SpinalHDL за пару-тройку ночей.

- Расчехлил анализатор сигналов Saleae Logic Pro 8.

- Приступил к изучению спецификации USB 1.0.

По факту на разработку ушло около 6 месяцев бурных ночей в обнимку с приборами.
Если бы знал заранее — не стал бы связываться.

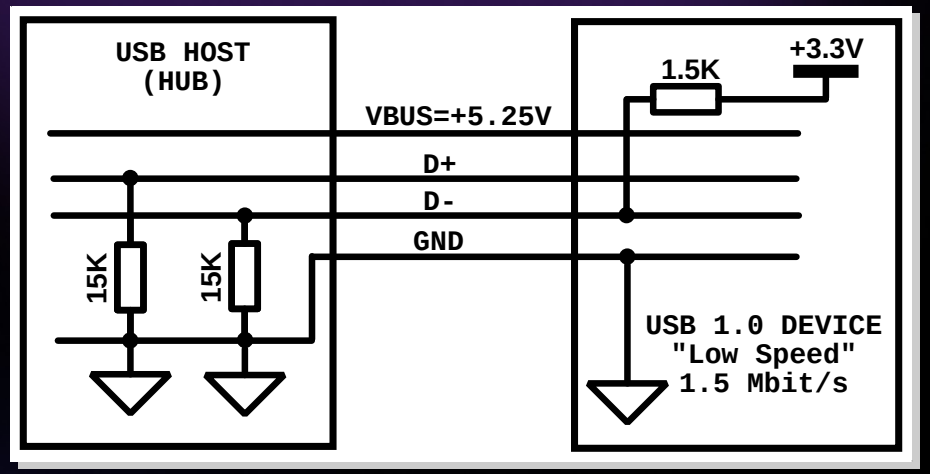
Saleae Logic 8 Pro →
Редкий зверь в наших краях

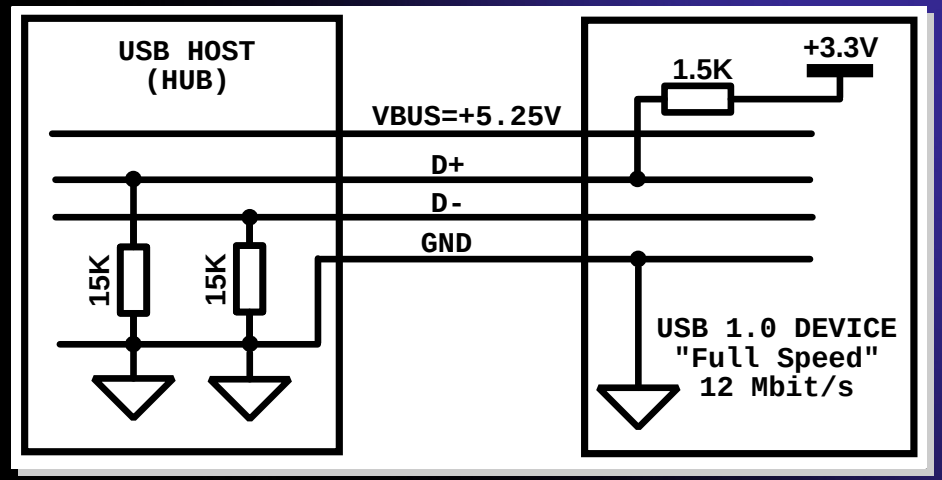
// II. Краткое описание стандарта USB 1.0

FPGA SYSTEMS
2025

3. USB 1.0 на физическом уровне
4. USB 1.0 на уровне представления данных
5. Виды транзакций USB 1.0
6. Древовидная структура конфигурационных данных
7. Процедура инициализация USB устройства (Enumeration)
8. USB Human Interface Device Class

3а. USB 1.0 на физическом уровне





Четыре возможных состояний шины представляются «символами»:

D+	D-	Символ для «Low Speed»	Символ для «Full Speed»
0 (GND)	0 (GND)	«SE0»	«SE0»
0 (GND)	1 (+3.3V)	«J»	«K»
1 (+3.3V)	0 (GND)	«K»	«J»
1 (+3.3V)	1 (+3.3V)	«SE1»	«SE1»

// 3b. Кодирование данных и сигналов

FPGA SYSTEMS
2025

Комбинации и последовательности символов:

Последовательность символов	Сигнал
«SE1»	Недопустимая комбинация (авария), требуется полный сброс
«J»	«Idle» - пассивное состояние
«SE0», «SE0», «J»	«EOP» - конец пакета или «KeepAlive» (каждые 1 мс)
«SE0» t>10мс	«Bus Reset» - полный сброс
«KJ KJ KJ KK»	«SYNC» - преамбула, начало пакета
«KJ» или «JK»	Передача бита «0»
«KK» или «JJ»	Передача бита «1»

Бит-стаффинг:

После каждых **6-т**и последовательных единиц (**«KKKKKK»** или **«JJJJJJ»**) передается один дополнительный ноль («J» или «K»)

- для устранения сбоев синхронизации и ошибок на приемной стороне.

// 4a. USB 1.0 на уровне представления данных

FPGA SYSTEMS 2025

- USB — шина с пакетной коммутацией в которой всем заправляет Хост.
- Три вида пакетов:

Короткий токен: ACK, NAK, STALL — 16 битовых интервалов + «EOP»

SYNC	PID ~PID	EOP
8	4 + 4	3
KJ KJ KJ KK	XXXX XXXX	SE0 SE0 J

Длинный токен: SETUP, IN, OUT — 32 битовых интервала + «EOP»

SYNC	PID ~PID	Device address (ADDR)	Endpoint (ENDP)	CRC5	EOP
8	8	7	4	5	3
KJ KJ KJ KK	XXXX XXXX	XXXX XXX	XXXX	XXXXX	SE0 SE0 J

Пакет с данными DATA0, DATA1 — от 32 до 96 битовых интервала + «EOP»

SYNC	PID ~DIP	Data	CRC16	EOP
8	8	64	16	3
KJ KJ KJ KK	XXXX XXXX	XXXX … XXXX	XXXX XXXX XXXX XXXX	SE0 SE0 J

// 4b. USB 1.0 на уровне представления данных: Packet ID

FPGA SYSTEMS
2025

USB 1.0 всего 8 типов пакетов (токенов):

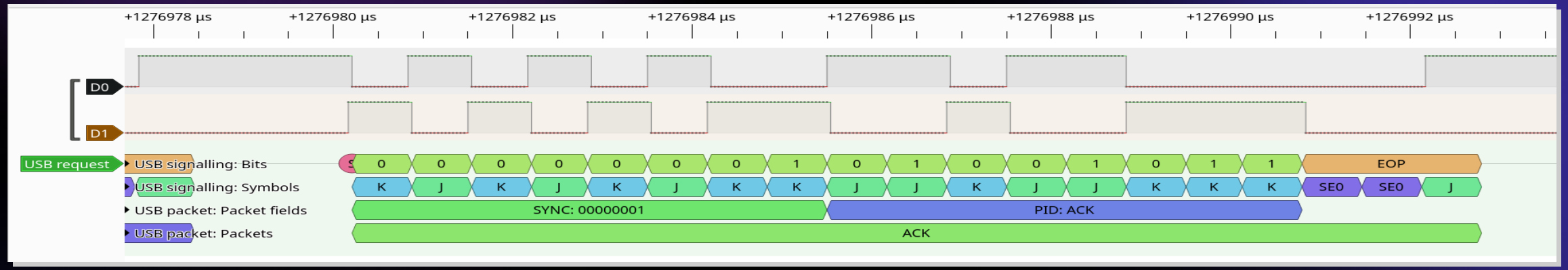
- 3 коротких токена;
- 3 длинных токена;
- 2 пакета для полезной нагрузки.

PID + DIP	Символы	Тип	Назначение
0100 1011	JJKJ JKKK	ACK	Подтверждение принятого пакета.
0101 1010	**JJKK KJJK**	**NAK**	**«Нет данных, приходите позже».**
0111 1000	JJJJ JKJK	STALL	Возникла неразрешимая ошибка.
1011 0100	KJJJ KKJK	SETUP	Служебный запрос к устройству. Задает ADDR и ENDP для зароса.
1001 0110	KJKK JJJK	IN	Инициация передачи **Device→Host**. Задает ADDR и ENDP для обмена.
1000 0111	KJKJ KKKK	OUT	Инициация передачи **Host->Device**. Задает ADDR и ENDP для обмена.
1100 0011	KKJK JKKK	DATA0	Четный пакет с данными.
1101 0010	KKJJ KJJK	DATA1	Нечётный пакет с данными.

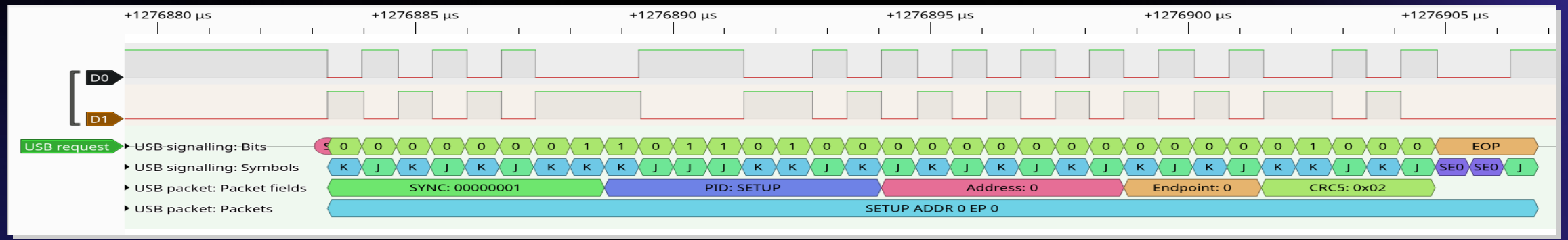
// 4c. Пакеты USB 1.0 на экране анализатора сигналов

FPGA SYSTEMS
2025

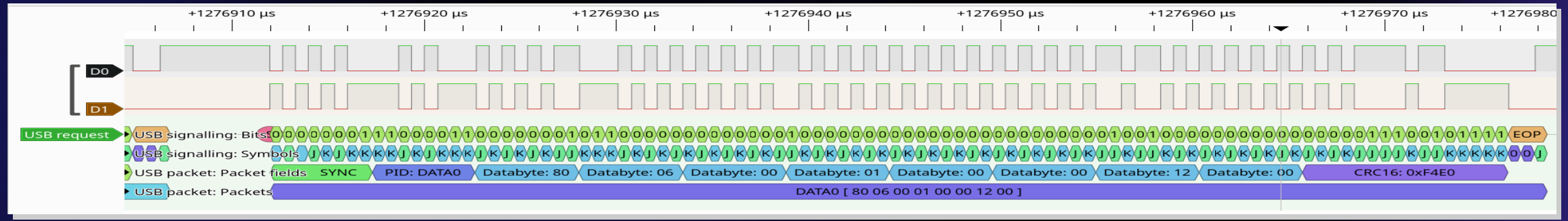
ACK



SETUP



DATA0



// 5a. Виды транзакций в USB 1.0

FPGA SYSTEMS
2025

В USB 1.0 выделяют следующие виды транзакций:

«**OUT Transfer**» — отправка данных в «пайп».
Host → Device

«**IN Transfer**» — получение данных от «пайпа».
Device → Host.

«**SETUP Request**» — служебный запрос, обычно передается в дефолтный «пайп».
Host → Device → ... → Host → Device.

«Пайп» - это комбинация из адреса устройства (ADDR) и номера конечной точки (ENDP).

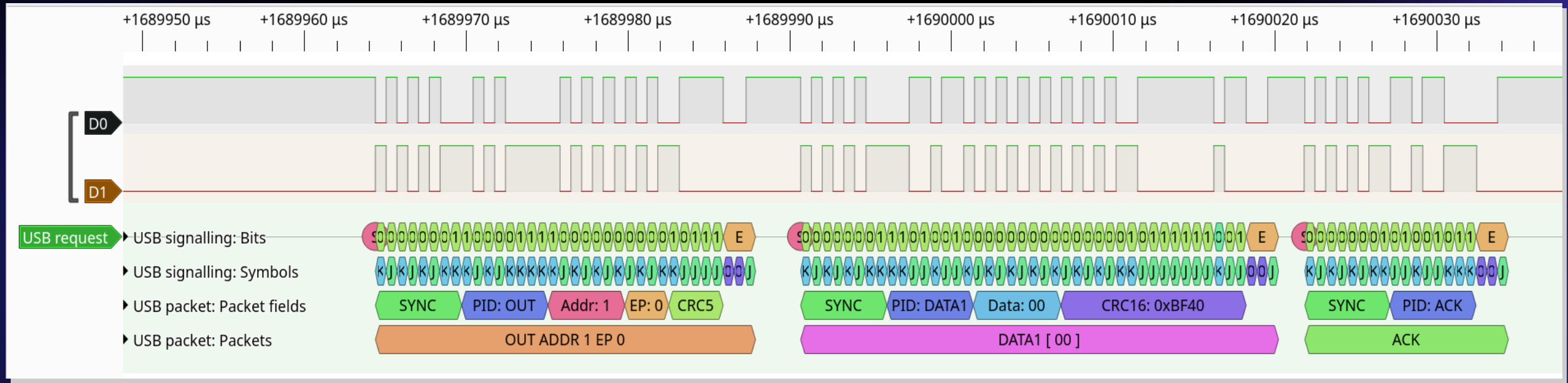
Каждое устройство имеет один «дефолтный пайп», ENDP = 0. Этот «пайп» используется для служебных нужд: запроса конфигурационных структур, установки параметров или считывания их текущих значений и состояний устройства.

// 5b. Транзакция «OUT Transfer»

FPGA SYSTEMS
2025

- Хост передает токен **OUT** с указанием ADDR:ENDP.
- Хост передает пакет с данными **DATAx** в течении 12 мкс.
- Устройство подтверждает прием в течении 12 мкс:
  ACK - «все ок, босс», **NAK** - «временно занят», или **STALL** - «что-то пошло не так».

- Если устройство не приняло пакет или приняло с ошибкой, то оно ничего не делает!

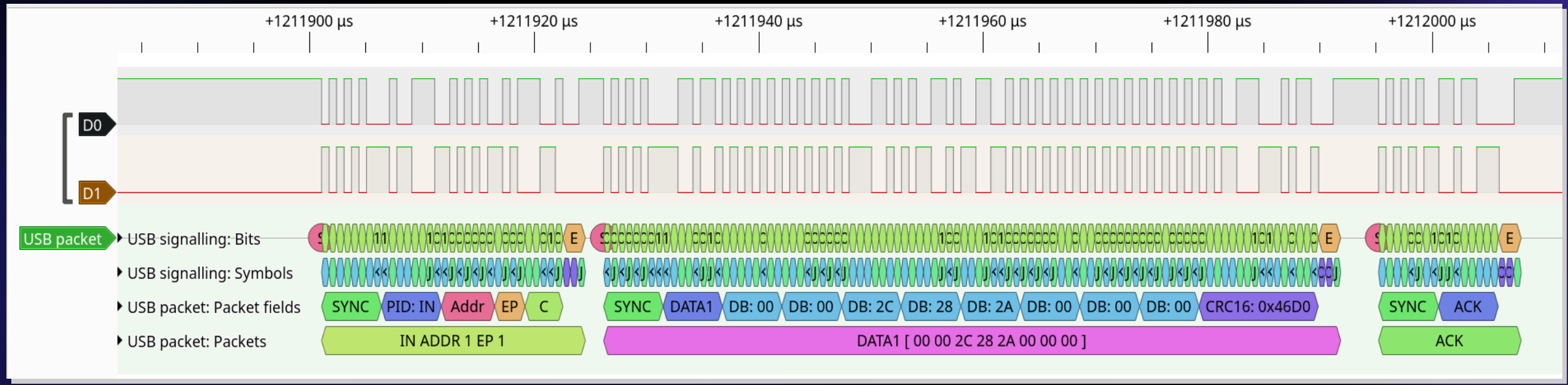


5c. Транзакция «IN Transfer»

FPGA SYSTEMS 2025

- Хост передает токен **IN** с указанием ADDR:ENDP.
- Устройство передает пакет с данными **DATAx** не позднее чем через 12 мкс.
- Хост подтверждает прием не позднее чем через 12 мкс:
  ACK - «принято» или **STALL** - «что-то пошло не так».

- Устройство может ответить **NAK** вместо пакета **DATAx**
  - это означает что у него сейчас нет данных для выдачи хосту.



// 5d. Транзакция «SETUP Request»

FPGA SYSTEMS
2025

Состоит из трех фаз:

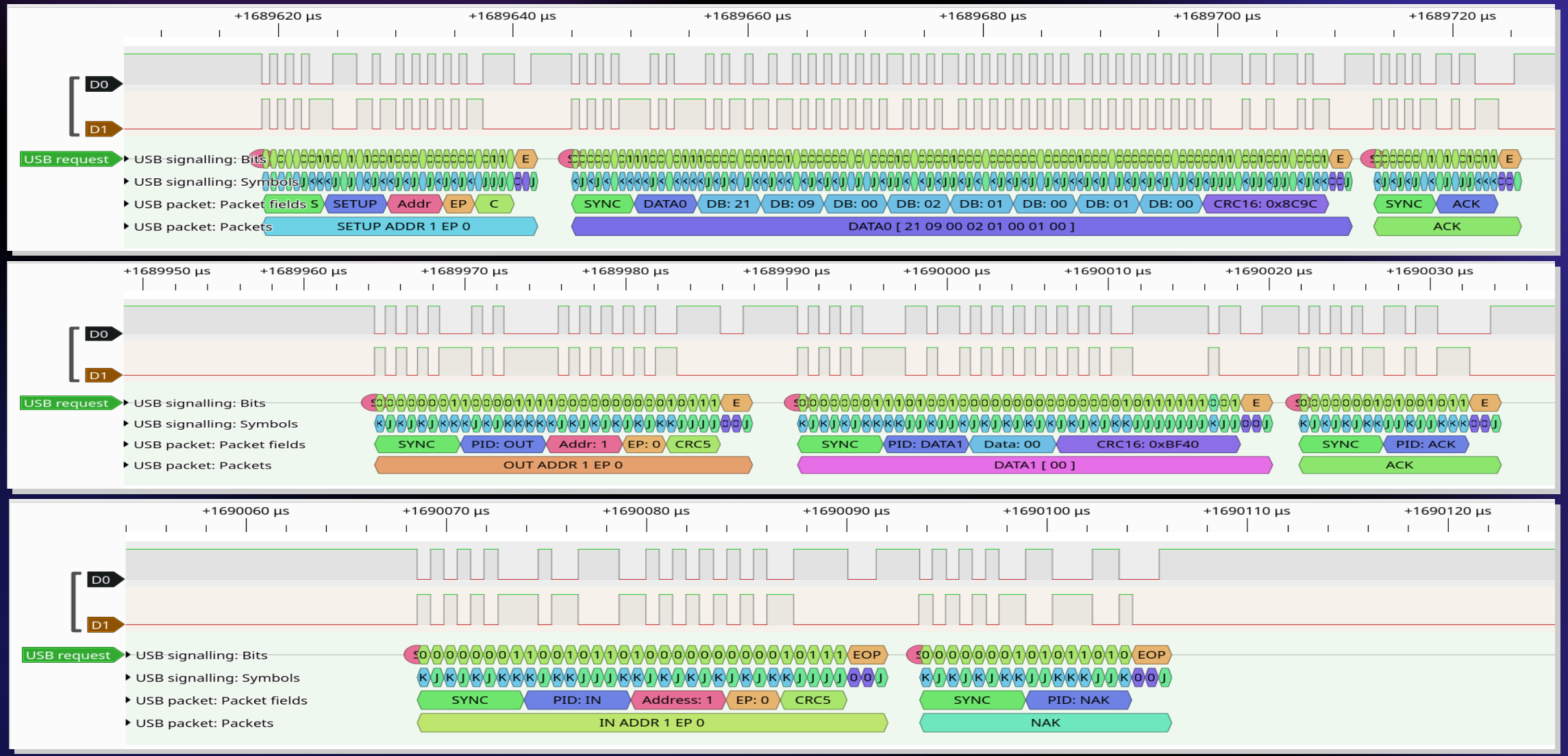
Фаза I. Установка соединения с «пайпом»:
- Хост передает токен **SETUP** с указанием ADDR:ENDP.
- Хост передает пакет **DATAx** содержащий данные запроса (всегда 8 байт).
- Устройство подтверждает прием:
  ACK - «все ок, босс» или **STALL** - «что-то пошло не так».

Фаза II. Обмен данными (передача, прием)
- Хост передает токен **OUT** (или **IN**) с указанием ADDR:ENDP.
- Хост передает (принимает) пакет с данными **DATAx**.
- Устройство подтверждает прием: **ACK**, **NAK**, **STALL**.

Фаза III: Разъединение.
- Хост передает токен IN (или OUT) с указанием ADDR:ENDP в попытке получить/передать данные от устройства.
- Устройство подтверждает факт отсутствия данных с помощью **NAK**.

5e. Транзакция «SETUP Request» на анализаторе

FPGA SYSTEMS
2025



// 5f. Виды «SETUP Request» запросов

FPGA SYSTEMS
2025

1. «Standard Device Request» — 8 запросов

bmRequestType	bRequest	wValue	wIndex	wLength	Возвр. данные
1000 0000b	GET_STATUS (0x00)	0	0	2	«Device Status»
0000 0000b	CLEAR_FEATURE (0x01)	Feature Selector	0	0	Нет
0000 0000b	SET_FEATURE (0x03)	Feature Selector	0	0	Нет
0000 0000b	SET_ADDRESS (0x05)	Device Address	0	0	Нет
1000 0000b	GET_DESCRIPTOR (0x06)	Descriptor Type & Index	0 или LANGID	Размер дескриптора	Запрашиваемый дескриптор
0000 0000b	SET_DESCRIPTOR (0x07)	Descriptor Type & Index	0 или LANGID	Размер дескриптора	Запрашиваемый дескриптор
1000 0000b	GET_CONFIGURATION (0x08)	0	0	1	Конфигурацион-ный дескриптор
0000 0000b	SET_CONFIGURATION (0x09)	Configur. Value	0	0	Нет

Запросы необходимы для работы со структурами данных внутри устройства и выполнения процедуры инициализации.

См. далее про «дескрипторы» →

2. «Standard Interface Request» - 5 запросов

bmRequestType	bRequest	wValue	wIndex	wLength	Возвр. данные
1000 0001b	GET_STATUS (0x00)	0	Interface	2	Два байта статуса - всегда нули!
0000 0001b	CLEAR_FEATURE (0x01)	Feature Selector	Interface	0	Нет
0000 0001b	SET_FEATURE (0x03)	Feature Selector	Interface	0	Нет
1000 0001b	GET_INTERFACE (0x0A)	0	Interface	1	Альтернативный интрефейс.
0000 0000b	SET_INTERFACE (0x0B)	Alt. Setting	Interface	0	Нет

3. «Standard Endpoint Request» - 4 запроса

bmRequestType	bRequest	wValue	wIndex	wLength	Возвр. данные
1000 0010b	GET_STATUS (0x00)	0	Endpoint	2	Два байта: бит 0 — Halt.
0000 0010b	CLEAR_FEATURE (0x01)	Feature Selector	Endpoint	0	Нет
0000 0010b	SET_FEATURE (0x03)	Feature Selector	Endpoint	0	Нет
1000 0010b	SYNCH_FRAME (0x0C)	0	Endpoint	Two	Номер кадра

// 6. Древовидная структура конфигурационных данных

FPGA SYSTEMS
2025

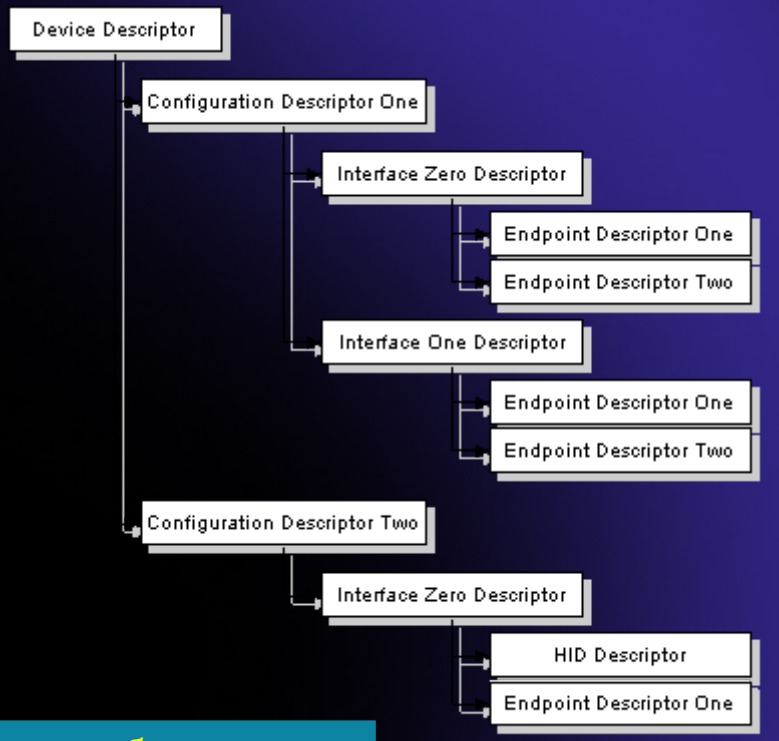
USB устройство содержит несколько структур данных (т. н. дескрипторов):

Стандартные дескрипторы:

«Device Descriptor» - общие данные об устройстве (VID, PID, Class).
«Configuration Descriptor» - наборы конфигурационных параметров.
«Interface Descriptor» - список интерфейсов и их параметров.
«Endpoint Descriptor» - список конечных точек и их параметров.

Расширения:

«Additional Descriptor» - всякая проприетарщина от производителя.
«HID Descriptor» - описывает формата данных возвращаемых USB HID устройством.



Все дескрипторы можно запросить командой:

Linux:

```
lsusb -vv
```

FreeBSD:

```
usbconfig -vv
```

Для того, чтобы начать работать с устройством его необходимо проинициализировать!

Для этого требуется вынуть из него всю эту древовидную структуру и выяснить количество интерфейсов, класс/подкласс, номера конечных точек и метод взаимодействия с ними!

// 7a. Процедура инициализация USB устройства (Enumeration)

FPGA SYSTEMS
2025

Обычно состоит из более чем 60-ти транзакций!

1. Первый сброс («Bus reset»).
2. Запрос сокращенной формы «Device Descriptor» и выяснения размера дерева структур и макс. Размер пакета.
3. Второй сброс («Bus reset»).
4. Присвоение адреса (SET_ADDRESS).
5. Второй запрос «Device Descriptor» - уже полный, со всеми подчиненными дескрипторами.
6. Установка необходимых параметров (SET_VALUE).
7. Активация конфигурационного профиля (SET_CONFIGURATION).
8. Анализ класса/подкласса и загрузка соответствующего драйвера.
    - запрос десткрипторов;
    - установка параметров;
    - и т. д.

Почему ДВА сброса ?

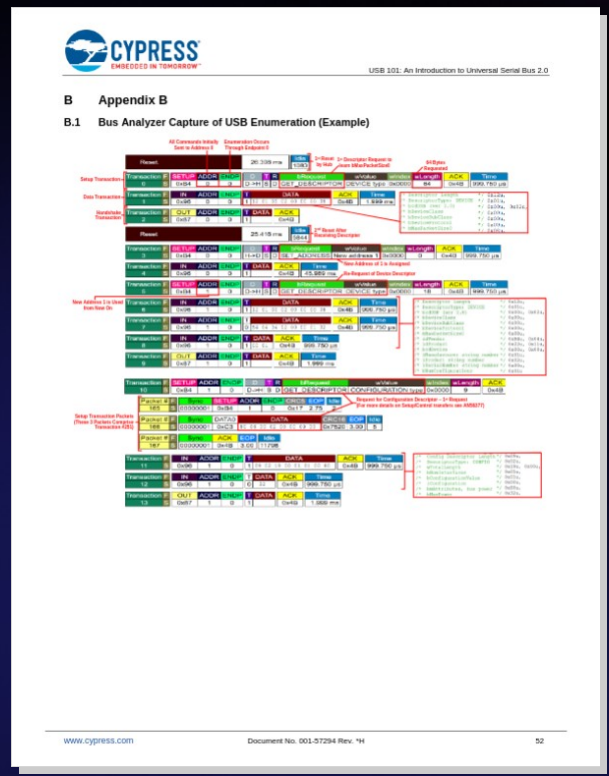
«...Additionally, the reason for performing a reset after only receiving 8 bytes is an artifact of early USB devices. In the early life of USB, some devices did not respond properly when a second request for the device descriptor occurred. To solve this problem, a reset after the first device descriptor request was required.»

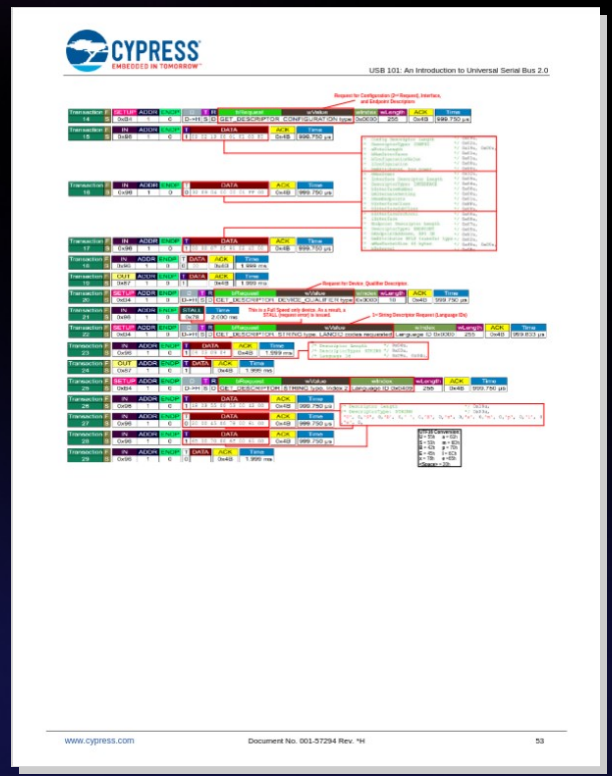
// 7b. Процедура инициализация USB устройства (Enumeration)

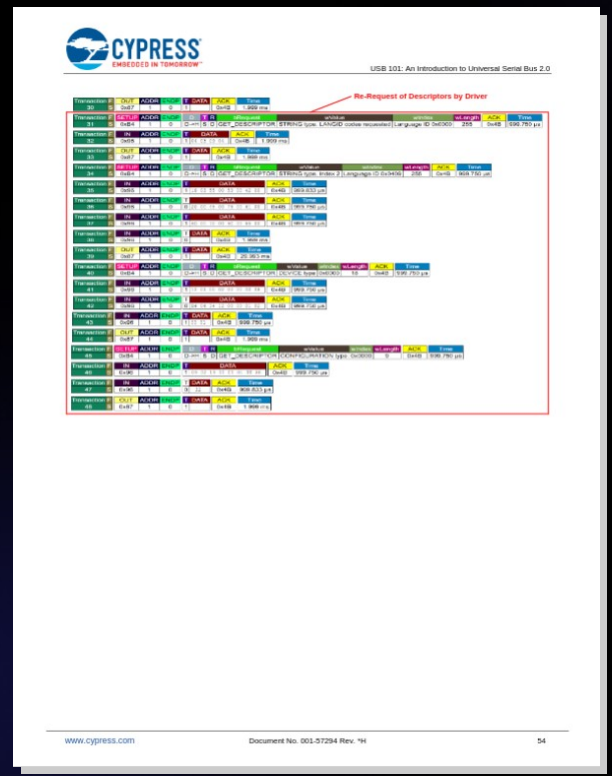
FPGA SYSTEMS

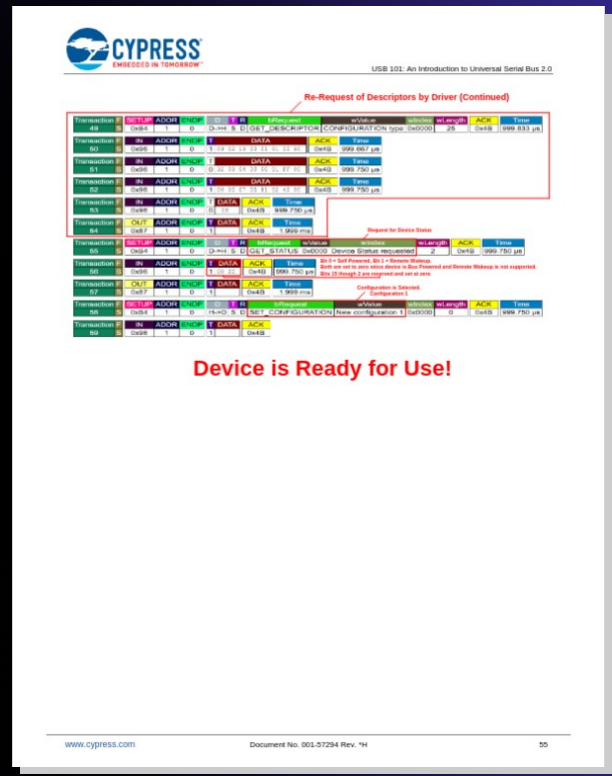
2025

AN57294: USB 101: An Introduction to Universal Serial Bus 2.0
от компании CYPRESS (см. «Appendix B»)









// 8a. USB Human Interface Device Class (HID)

FPGA SYSTEMS
2025

USB устройства бывают различных классов (HID, Serial, Storage, etc):

- позволяет создавать унифицированные драйверы независимые от VID/PID.

Спецификация USB HID Class описывает устройства ввода:

- клавишные («keyboard» и «keypad»);
- курсорные манипуляторы («мышь», «track ball», etc);
- рычажные («joystick», «gamepad», «rudder»);
- сенсорные панели и т. д.

Два протокола:

- «**Report Protocol**»
  - описывает очень сложный универсальный формат представления данных и событий;
  - требуется отдельная VM для интерпретации данных;
  - необходимо парсить очень сложную структуру «HID Report Descriptor».

- «**Boot Protocol**» (Appendix B спецификации HID)
  - все данные и события находятся в одном DATA пакете длиной не более 8 байт;
  - одна транзакция типа «IN Transfer» на ENDP = 1;
  - расположение полей данных имеет фиксированные позиции в пакете (кроме клавиатуры).

«Boot Protocol» придуман для упрощения реализации поддержки HID в ранних BIOS-ах на IBM PC совместимых машинах. Сейчас используется повсеместно.

Драйвер USB HID в Unix системах поддерживает только «Boot Protocol».

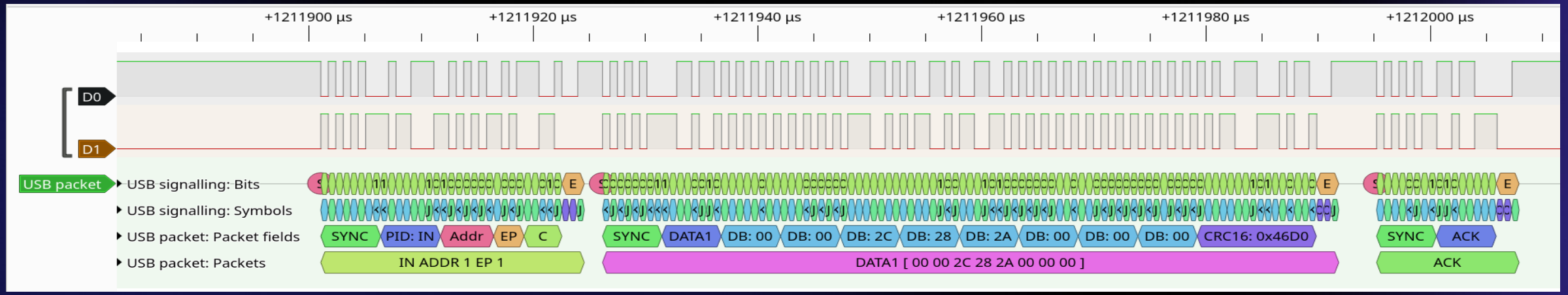
// 8b. Формат данных USB HID для кнопочных устройств

FPGA SYSTEMS
2025

Формат пакета с событиями от клавиатуры согласно Appendix B.1. («Boot Protocol»):

Номера битов:	63 … 56	55 … 48	47 … 40	39 … 32	31 … 24	23 … 16	15 … 8	7 … 0
Назначение:	UsageID	UsageID	UsageID	UsageID	UsageID	UsageID	Reserved	Bitmap
Пример:	**0x00**	**0x00**	**0x00**	**0x2A**	**0x28**	**0x2C**	**0x00**	**0x00**
Описание:	Нет данных	Нет данных	Нет данных	BACKSPACE	ENTER	SPACEBAR	Резерв	Нет модификаторов

Транзакция «IN Transfer» для получение данных от клавиатуры согласно Appendix B.1. («Boot Protocol»):



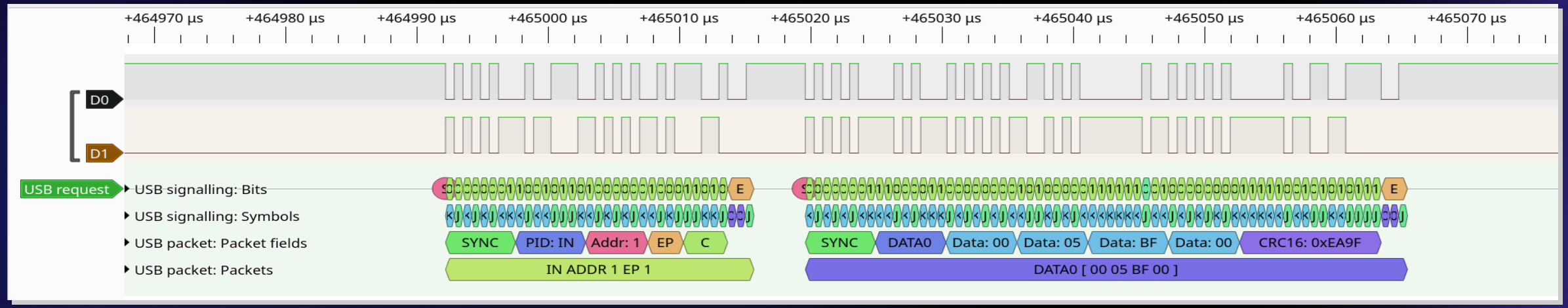
8c. Формат данных USB HID для указательных устройств

FPGA SYSTEMS
2025

Формат пакета с событиями от манипулятора «мышь» согласно Appendix B.2. («Boot Protocol»):

Номера битов:	**31 … 24**	**23 … 16**	**15 … 8**	**7 … 0**
Назначение:	(Optional)	UsageID	UsageID	Bitmap
Пример:	**0x00**	**0xBF**	**0x05**	**0x00**
Описание:	Состояние «колеса» не менялось	dY=-41	dX = +5	Кнопки не нажаты

Транзакция «IN Transfer» для получение данных от манипулятора «мышь» согласно Appendix B.2. («Boot Protocol»):



// III. Описание КА USB 1.0 хост-контроллера

FPGA SYSTEMS
2025

9. Требования к USB хост-контроллеру
10. Структура USB хост-контроллера
11. Основной КА USBMain
12. КА USBSendSE0 для выполнения сброса и «KeepAlive»
13. КА USBSendShortToken для передачи короткого токена
14. КА USBSendLongToken для передачи длинного токена
15. КА USBSendData для передачи пакета данных DATAx
16. КА USBReceiver для приема произвольного пакета данных
17. Реализация КА хост-контроллера на SpinalHDL
18. Реализация основного КА и логики управления
19. Интеграция хост-контроллера в СнК
20. Сборка проекта (синтез битстрима)

// 9a. Требования к USB хост-контроллеру

FPGA SYSTEMS
2025

В процессе изучения существующих хост-контроллеров (UHCI, OHCI, EHCI), выкристаллизовались следующие требования:

- Содержать минимум аппаратуры достаточный для осуществления обмена по шине.
- Самостоятельно вычислять CRC5 и CRC16 для переданных пакетов.
- Самостоятельно принимать пакет произвольной длины вычисляя и сверяя CRC16.
- Самостоятельно выполнять процедуру «KeepAlive» каждые 1 мс.
- Вся логика выполнения транзакций возлагается на программную часть.

ПО взаимодействует с хост-контоллером через регистры отображаемые в память:

- ПО посылает команды контроллеру через регистр команд и устанавливает флаг «на старт»;
- ПО ожидает окончания исполнения считывая флаг «готово» в регистре статуса.
- ПО может ожидать прерывания или опрашивать дополнительные флаги регистра статуса:
    - «авария на шине» - устройство отключено или не отвечает;
    - «контроллера занят» - выполняет сброс, «keepalive», передает или принимает данные.
    - «команда исполнена» - сбрасывается перед началом выполнения и устанавливается по готовности;
    - «принят пакет от устройства»;
    - «CRC16 корректный» - для принятого пакета.
- ПО может считывать текущее состояние основного КА - в целях отладки.

// 9b. Требования к USB хост-контроллеру

FPGA SYSTEMS
2025

Хост-контроллер должен формировать прерывания по следующим событиям:

- «команда исполнена»;
- «принят пакет данных»;
- «ошибка на шине»;
- «обнаружено подключение к шине».

Команды исполняемые хост-контроллером:

- «**CMDBusReset**» - выполнить сброс шины;
- «**CMDSendShortToken**» - переслать короткий токен (ACK, NAK, STALL);
- «**CMDSendLongToken**» - переслать длинный токен (IN, OUT, SETUP) вычислив и подставив CRC5;
- «**CMDSendData**» - переслать пакет с данными (DATA0, DATA1) вычислив и подставив CRC16.

// 10. Структура USB хост-контроллера

FPGA SYSTEMS
2025

- Один основной КА **USBMain** — мониторит состояние шины, исполняет команды путем активирования оного из вспомогательных КА, формирует флаги и прерывания, транслирует результат работы вспомогательных КА пользователю.

- Пять вспомогательных КА:

USBSendSE0 — Переводит USB шину в состояние «SE0» на заданное число битовых интервалов. Может быть использован как для посылки «KeepAlive», так и для сброса шины («Bus reset»).

USBSendShortToken — Посылает короткий токен (ACK, NAK, STALL).

USBSendLongToken — Посылает длинный токен (IN, OUT, SETUP) на заданный ADDR:ENDP. Вычисляет CRC5 и подставляет в пакет.

USBSendData — Посылает токен DATAx вместе с данными (до 64 бит), вычисляет CRC16 и подставляет в пакет.

USBReceiver — Принимает пакет произвольной длины, вычисляет и проверяет CRC16 .

- Все вспомогательные КА имеют общий базовый I/O интерфейс описываемый классом **USB_IO**.

- Все вспомогательные КА активируются по сигналу **io.valid** и формируют сигнал готовности **io.ready**.

- При активации вспомог. КА производится коммутация шины.

- Интерфейсный класс USB_IO содержит делитель частоты **clock_div** для вычисления длительности битового интервала.

```
// Шина USB
class USBInterface() extends Bundle with IMasterSlave{
  val dm = inout(Analog(Bool()))
  val dp = inout(Analog(Bool()))
}

// Базовый интерфейсный класс
class USB_IO extends Bundle {
    val usb       = USBInterface()
    val valid     = in Bool()
    val ready     = out Bool()
    val clock_div = in UInt(8 bits)
    val test      = out Bool()

    ready := False // default value, it will be changed in implementation
}
```

// 11. Основной КА USBMain

FPGA SYSTEMS
2025

Состояния основного КА **USBMain:**

StateUnconnected —
Начальное состояние, на шине нет подключения;

StateWaitCMDorSYNC —
Бездействует, готов исполнить команду или принять пакет.

StateSendReset —
Выполняет процедуру «Bus reset», активен КА **USBSendSE0**;

StateKeepAlive —
Выполняет процедуру «KeepAlive», активен КА **USBSendSE0**;

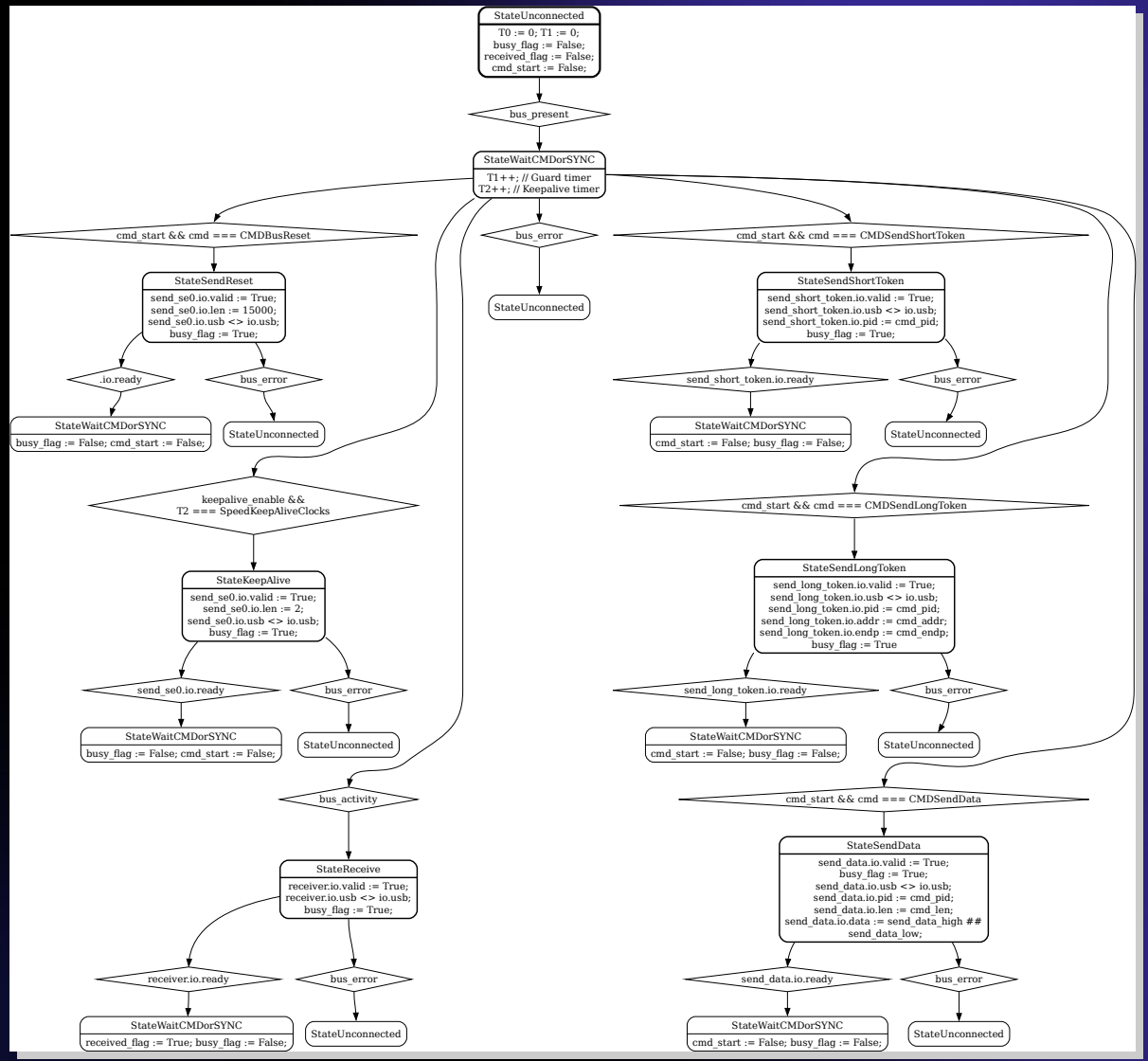
StateSendShortToken —
Передает короткий токен активен КА **USBSendShortToken**;

StateSendLongToken —
Передает длинный токен активен КА **USBSendLongToken**;

StateSendData —
Передает DATA пакет, активен КА **USBSendData**;

StateReceive —
Принимает пакет, активен КА **USBRecever**;

Получено с помощью **GraphViz** →



// 12. КА USBSendSE0 для выполнения сброса и «KeepAlive»

FPGA SYSTEMS
2025

Вспомогательный КА **USBSendSE0** удерживает шину USB в состоянии «SE0» в течении заданного количества битовых интервалов.

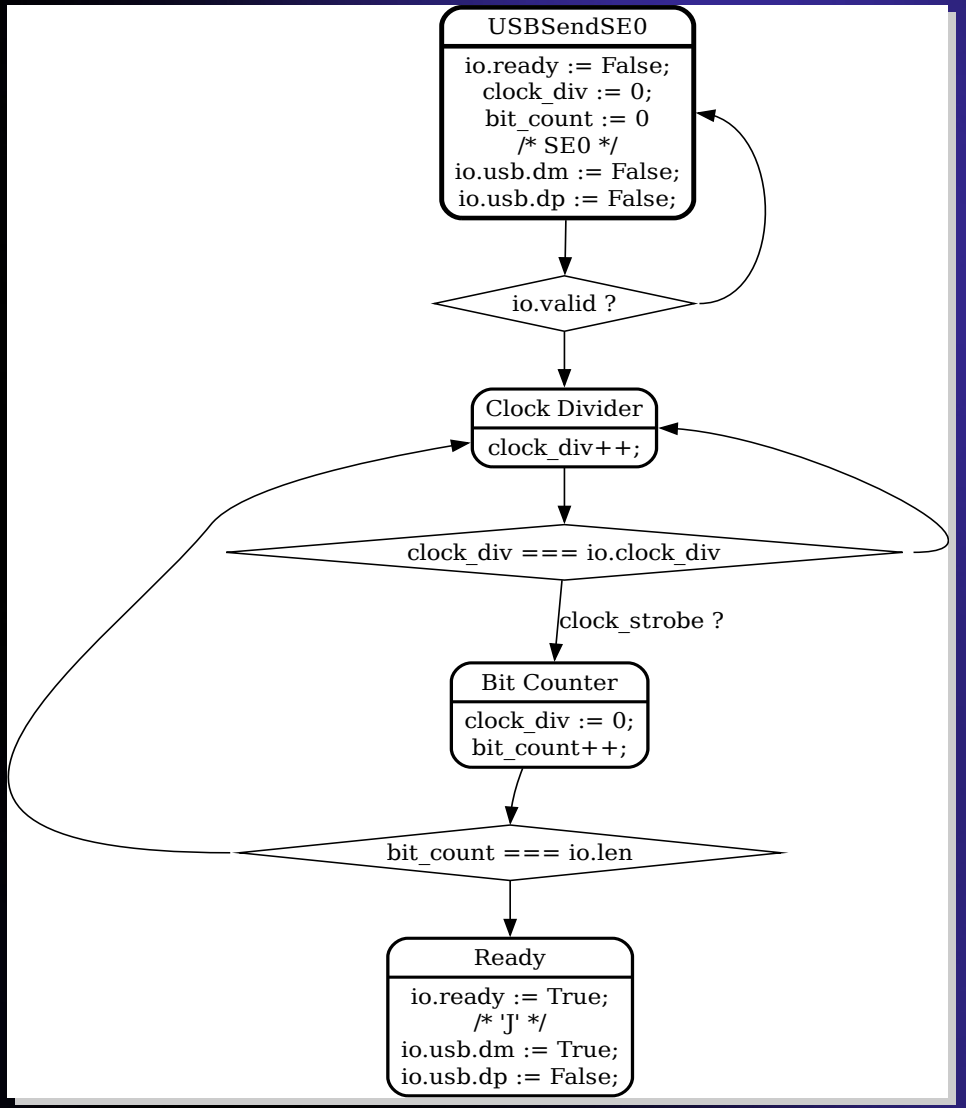
- активируется по сигналу **io.valid**;

- устанавливает **io.usb.dm** и **io.usb.dp** в значение **False** (подтягивает к «земле»);

- вычисляет длительность одного битового интервала исходя из входного значения **io.clock_div** и формирует внутренний строб **clock_strobe**;

- считает число стробов (битовых интервалов) в регистре **bit_count**;

- достигнув значения **io.len** завершает работу возвращая шину в состояние **«J»**, при этом устанавливает флаг формируя **io.ready**.



// 13a. КА USBSendShortToken для передачи короткого токена

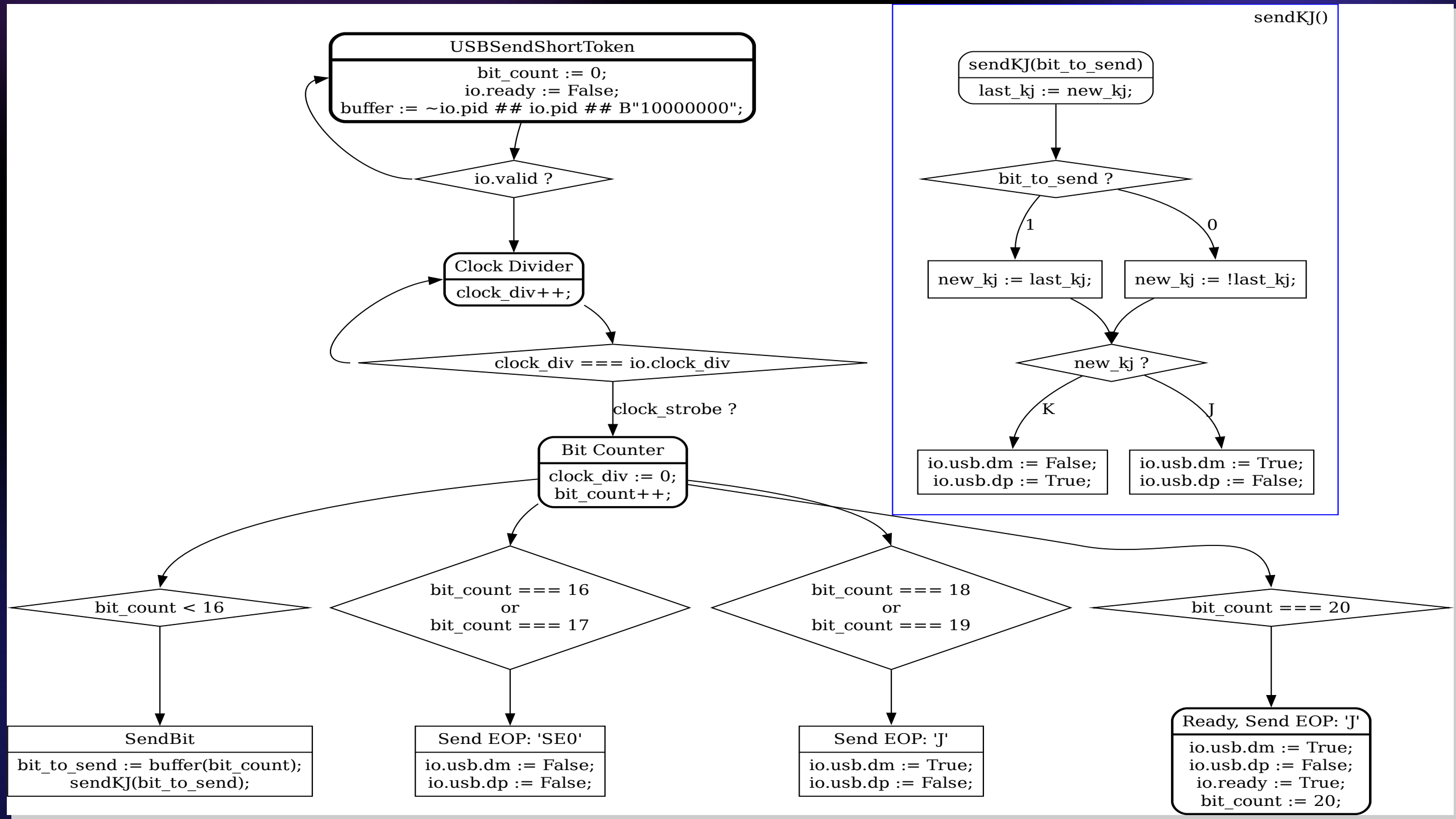
FPGA SYSTEMS 2025

Вспомогательный КА **USBSendShortToken** передает короткий токен (ACK, NAK, STALL)

- активируется по сигналу **io.valid**;
- формирует битовую последовательность **SYNC ## PID ## ~PID** в одном 16 битном регистре **buffer**;
- вычисляет длительность одного битового интервала исходя из входного значения **io.clock_div** и формирует внутренний строб **clock_strobe**;
- по каждому стробу передает на шину один «символ» извлекая бит из **buffer** по индексу **bit_count** и преобразуя в «символ» с помощью процедуры **sendKJ()**;
- считает число стробов (битовых интервалов) в регистре **bit_count**;
- передав **16 бит** переходит в состояние передачи **EOP** (конец пакета: «SE0», «SE0», «J», «J», ready).

Содержит вложенный автомат **sendKJ()** который преобразует бит данных в «символ» исходя из последнего переданного «символа» запоминаемого в регистре **last_kj** и текущего передаваемого бита данных в **buffer(bit_count)**;

диаграмма КА на следующем слайде →


USBSendShortToken
bit_count := 0;
io.ready := False;
buffer := ~io.pid ## io.pid ## B"10000000";
io.valid ?
Clock Divider
clock_div++;
clock_div === io.clock_div
clock_strobe ?
Bit Counter
clock_div := 0;
bit_count++;
bit_count < 16
SendBit
bit_to_send := buffer(bit_count);
sendKJ(bit_to_send);
bit_count === 16
or
bit_count === 17
Send EOP: 'SE0'
io.usb.dm := False;
io.usb.dp := False;
bit_count === 18
or
bit_count === 19
Send EOP: 'J'
io.usb.dm := True;
io.usb.dp := False;
bit_count === 20
Ready, Send EOP: 'J'
io.usb.dm := True;
io.usb.dp := False;
io.ready := True;
bit_count := 20;
sendKJ()
sendKJ(bit_to_send)
last_kj := new_kj;
bit_to_send ?
1
0
new_kj := last_kj;
new_kj := !last_kj;
new_kj ?
K
J
io.usb.dm := False;
io.usb.dp := True;
io.usb.dm := True;
io.usb.dp := False;

// 14a. КА USBSendLongToken для передачи длинного токена

FPGA SYSTEMS
2025

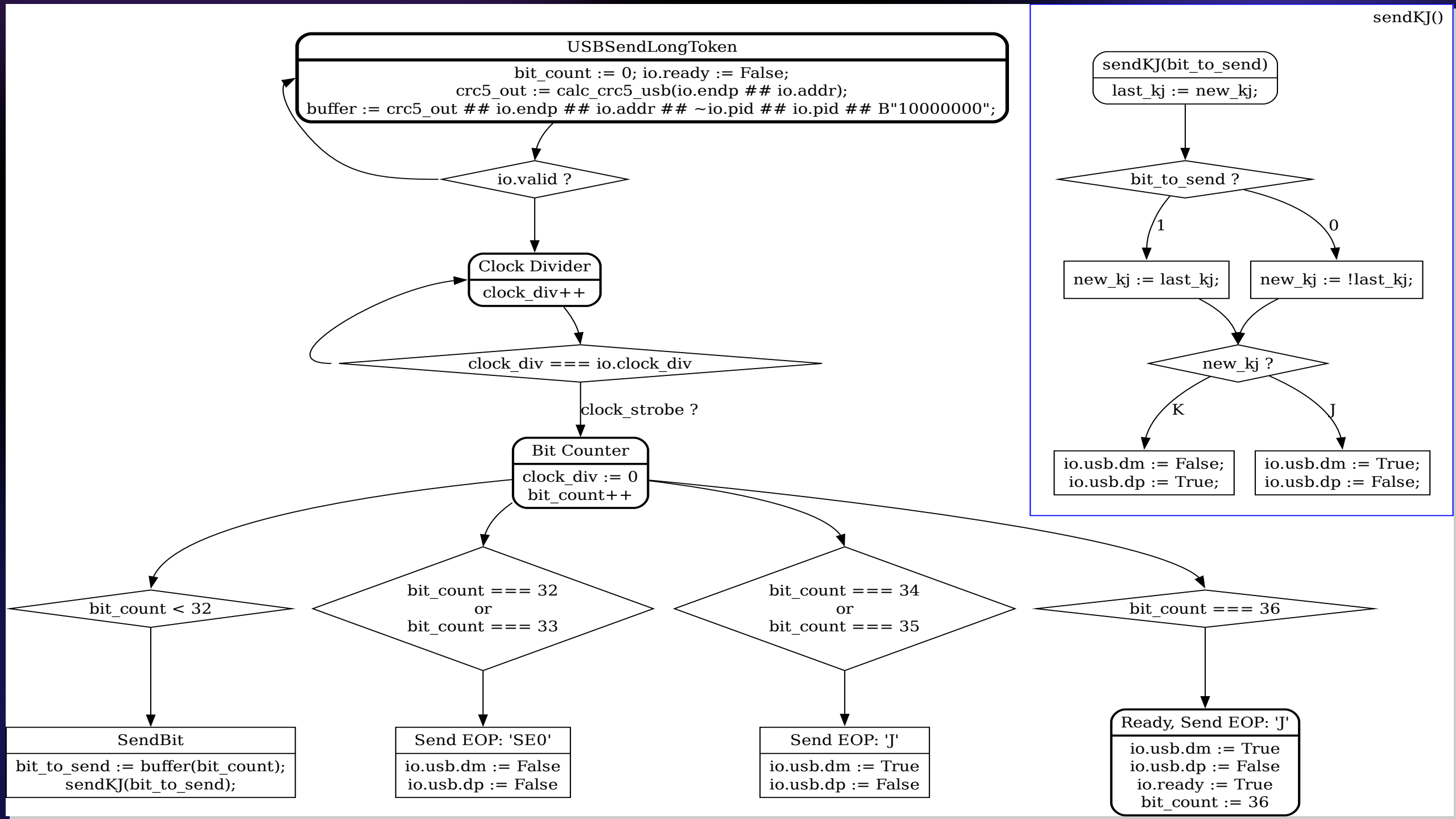
Вспомогательный КА **USBSendLongToken** передает длинный токен (IN, OUT, SETUP)

- активируется по сигналу **io.valid;**
- формирует битовую последовательность **SYNC ## PID ## ~PID ## ADDR ## ENDP ## CRC5** в одном **32** битном регистре **buffer**;
- вычисляет длительность одного битового интервала исходя из входного значения **io.clock_div** и формирует внутренний строб **clock_strobe**;
- по каждому стробу передает на шину один «символ» извлекая бит из **buffer** по индексу **bit_count** и преобразуя в «символ» с помощью **sendKJ()**;
- считает число стробов (битовых интервалов) в регистре **bit_count;**
- передав **32** бита переходит в состояние передачи **EOP** (конец пакета: «SE0», «SE0», «J», «J», ready).

Содержит вложенный автомат **sendKJ()** для преобразования передаваемого бита данных извлекаемого из **buffer(bit_count)** в «символ».

Содержит функцию **calc_crc5_usb()** для вычисления код CRC5.
Функция порождает комбинационную логику.

диаграмма КА на следующем слайде →


USBSendLongToken
bit_count := 0; io.ready := False;
crc5_out := calc_crc5_usb(io.endp ## io.addr);
buffer := crc5_out ## io.endp ## io.addr ## ~io.pid ## io.pid ## B"10000000";
io.valid ?
Clock Divider
clock_div++
clock_div === io.clock_div
clock_strobe ?
Bit Counter
clock_div := 0
bit_count++
bit_count < 32
bit_count === 32
or
bit_count === 33
bit_count === 34
or
bit_count === 35
bit_count === 36
SendBit
bit_to_send := buffer(bit_count);
sendKJ(bit_to_send);
Send EOP: 'SE0'
io.usb.dm := False
io.usb.dp := False
Send EOP: 'J'
io.usb.dm := True
io.usb.dp := False
Ready, Send EOP: 'J'
io.usb.dm := True
io.usb.dp := False
io.ready := True
bit_count := 36
sendKJ()
sendKJ(bit_to_send)
last_kj := new_kj;
bit_to_send ?
1
0
new_kj := last_kj;
new_kj := !last_kj;
new_kj ?
K
J
io.usb.dm := False;
io.usb.dp := True;
io.usb.dm := True;
io.usb.dp := False;

// 15a. КА USBSendData для передачи пакета данных DATAx

FPGA SYSTEMS 2025

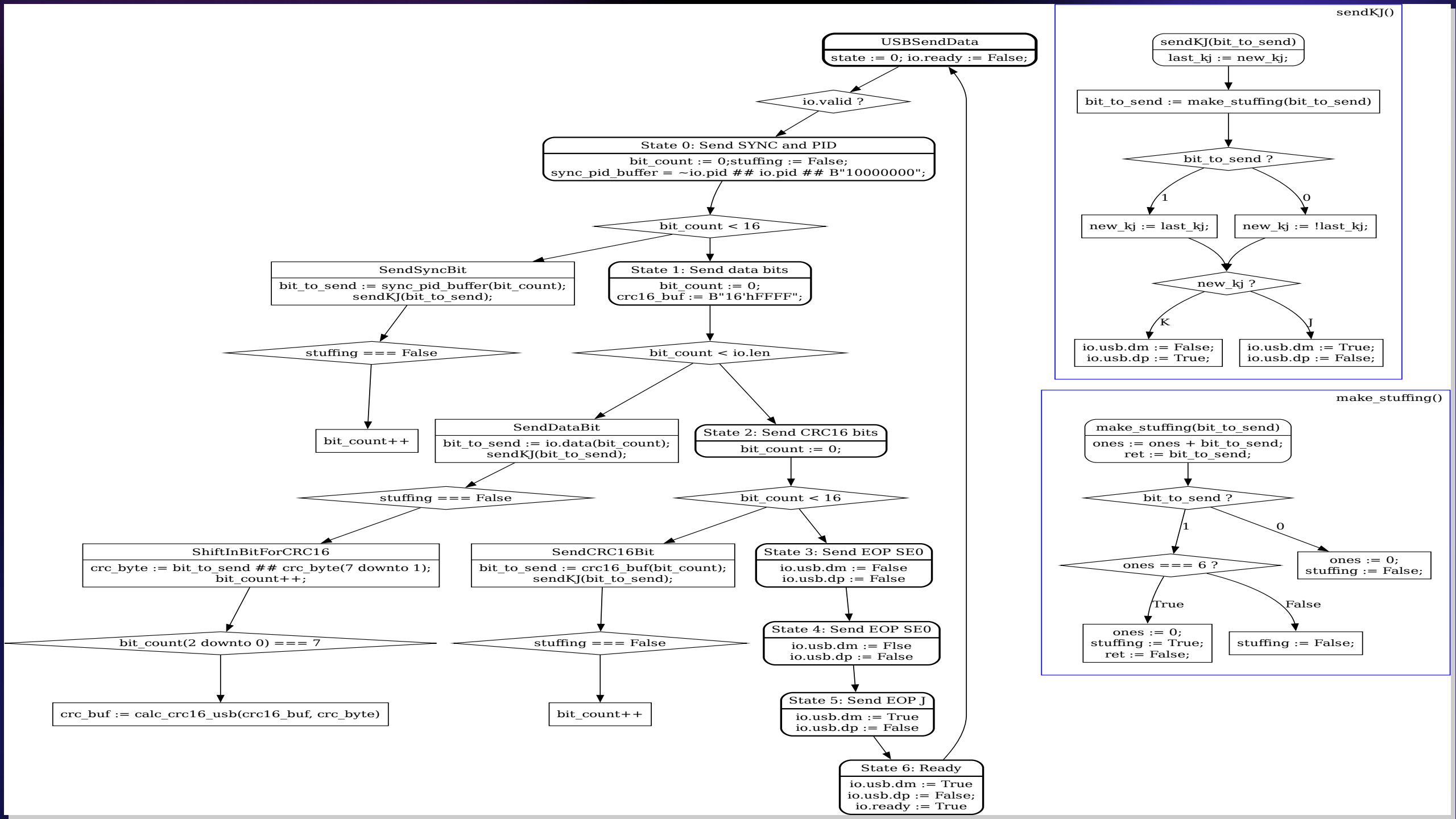
Вспомогательный КА **USBSendData** передает пакет с данными (DATA0, DATA1) длиной до **64** бит.

- Активируется по сигналу **io.valid**, вычисляет строб **clock_strobe** для битового интервала.
- Содержит вложенный автомат **sendKJ()** для преобразования передаваемого бита данных в «символ».
- Содержит вложенный автомат **make_stuffing()** который учитывает число переданных единиц в регистре **ones** и поднимает флаг **stuffing** если **ones === 6**.
- Выполняет **бит-стаффинг** — передает дополнительный ноль если флаг **stuffing** установлен.
- Содержит функцию **calc_crc16_usb()** для по-байтового вычисления код CRC16, порождает комбинационную логику.

- Работает в **пять фаз**, на каждой фазе свой буфер исходных данных:

Фаза 1: передается преамбула и PID (**SYNC ## PID ## ~PID**) — 16 битовых интервалов;
Фаза 2: передается блока данных из входного буфера **io.data** длиной **io.len** бит;
попутно вычисляется CRC16 для каждых 8 бит переданных данных (по-байтово);
Фаза 3: передается вычисленный код CRC16 из регистра **crc16_buf** — 16 бит;
Фаза 4: передается **EOP** — «SE0», «SE0», «J», «J»;
Фаза 5: останов и подача флага **io.ready**.

диаграмма КА на следующем слайде →


USBSendData
state := 0; io.ready := False;
io.valid ?
State 0: Send SYNC and PID
bit_count := 0;stuffing := False;
sync_pid_buffer = ~io.pid ## io.pid ## B"10000000";
bit_count < 16
SendSyncBit
bit_to_send := sync_pid_buffer(bit_count);
sendKJ(bit_to_send);
State 1: Send data bits
bit_count := 0;
crc16_buf := B"16'hFFFF";
stuffing === False
bit_count < io.len
bit_count++
SendDataBit
bit_to_send := io.data(bit_count);
sendKJ(bit_to_send);
State 2: Send CRC16 bits
bit_count := 0;
stuffing === False
bit_count < 16
ShiftInBitForCRC16
crc_byte := bit_to_send ## crc_byte(7 downto 1);
bit_count++;
SendCRC16Bit
bit_to_send := crc16_buf(bit_count);
sendKJ(bit_to_send);
State 3: Send EOP SE0
io.usb.dm := False
io.usb.dp := False
bit_count(2 downto 0) === 7
stuffing === False
State 4: Send EOP SE0
io.usb.dm := Flse
io.usb.dp := False
crc_buf := calc_crc16_usb(crc16_buf, crc_byte)
bit_count++
State 5: Send EOP J
io.usb.dm := True
io.usb.dp := False
State 6: Ready
io.usb.dm := True
io.usb.dp := False;
io.ready := True
sendKJ()
sendKJ(bit_to_send)
last_kj := new_kj;
bit_to_send := make_stuffing(bit_to_send)
bit_to_send ?
1
0
new_kj := last_kj;
new_kj := !last_kj;
new_kj ?
K
J
io.usb.dm := False;
io.usb.dp := True;
io.usb.dm := True;
io.usb.dp := False;
make_stuffing()
make_stuffing(bit_to_send)
ones := ones + bit_to_send;
ret := bit_to_send;
bit_to_send ?
1
0
ones === 6 ?
ones := 0;
stuffing := False;
True
False
ones := 0;
stuffing := True;
ret := False;
stuffing := False;

// 16a. КА USBReceiver для приема произвольного пакета данных

FPGA SYSTEMS 2025

Вспомогательный КА **USBReceiver** принимает от устройства пакет произвольной длины до 127 бит: (PID + ~PID + DATA + CRC16).

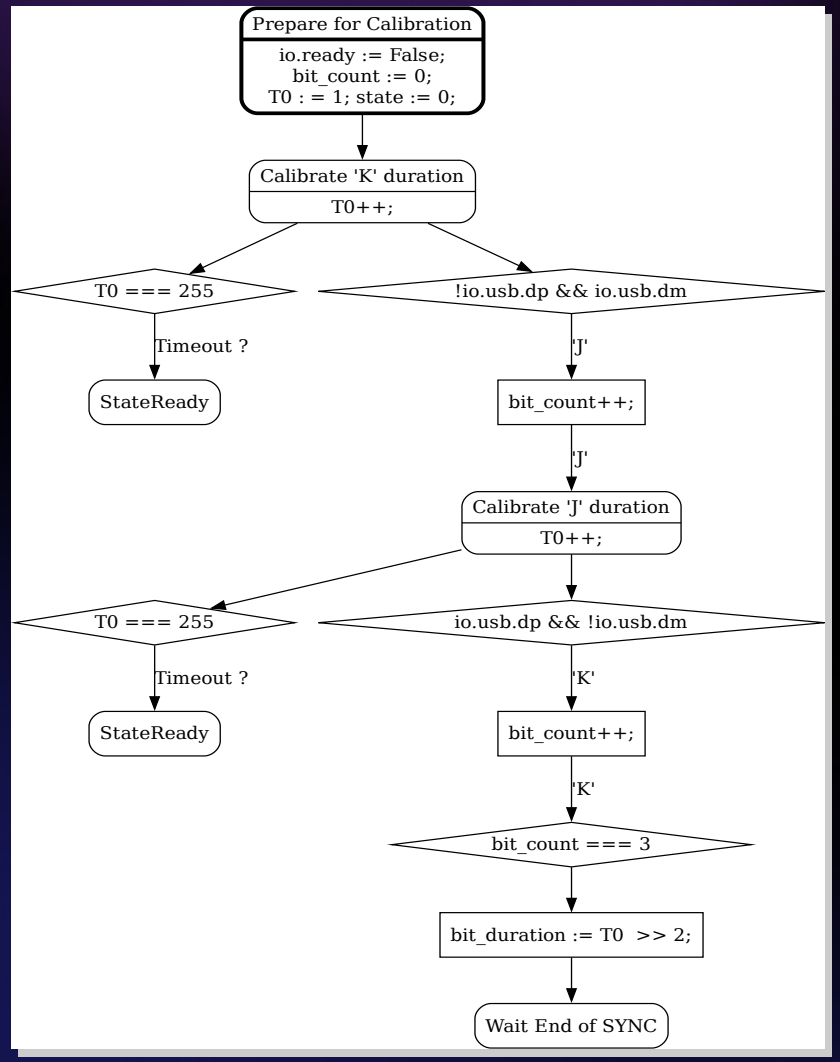
- Активируется по сигналу **io.valid** в момент обнаружения активности на шине («J» → «K»).
- Выполняет **бит-де-стаффинг** — удаляет дополнительный ноль если счетчик единиц **ones === 6**.
- Содержит функцию **calc_crc16_usb()** для по-байтового вычисления код CRC16 для принимаемых данных.
- Работает в **три фазы**:

Фаза 1: Вычисление (калибровка) длительности битового интервала **bit_duration** по первым символам преамбулы («K», «J», «K», «J»).
Фаза 2: Ожидание конца преамбулы (пропустить «K», «K»).
Фаза 3: Прием данных в выходной буфер **io.packet** длиной 128 бит;
- вычисляется CRC16 для каждых 8 бит принятых данных (по-байтово);
- на каждом фронте и спаде производится пере-синхронизация;
- если принято состояние **«SE0»** или **bit_count === 127**, то завершение и подъем **io.ready**.
- если сработал защитный таймер, то завершение и подъем **io.ready**;

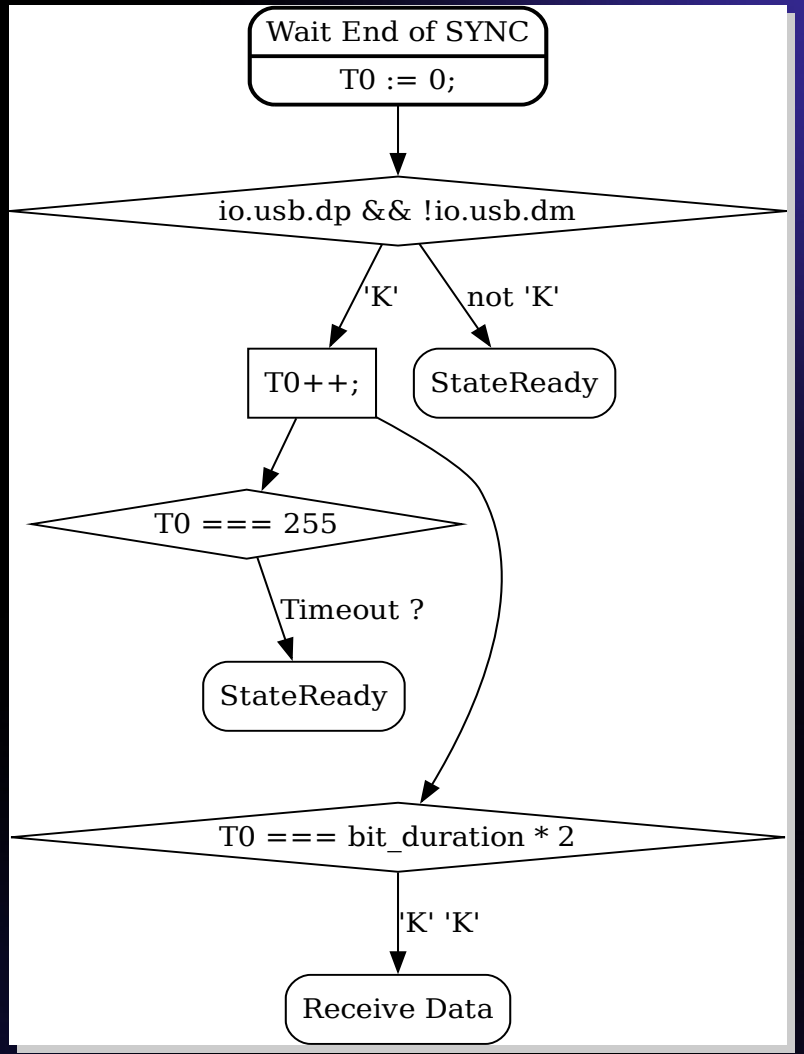
диаграмма КА на следующем слайде →

16b. КА USBReceiver для приема произвольного пакета данных

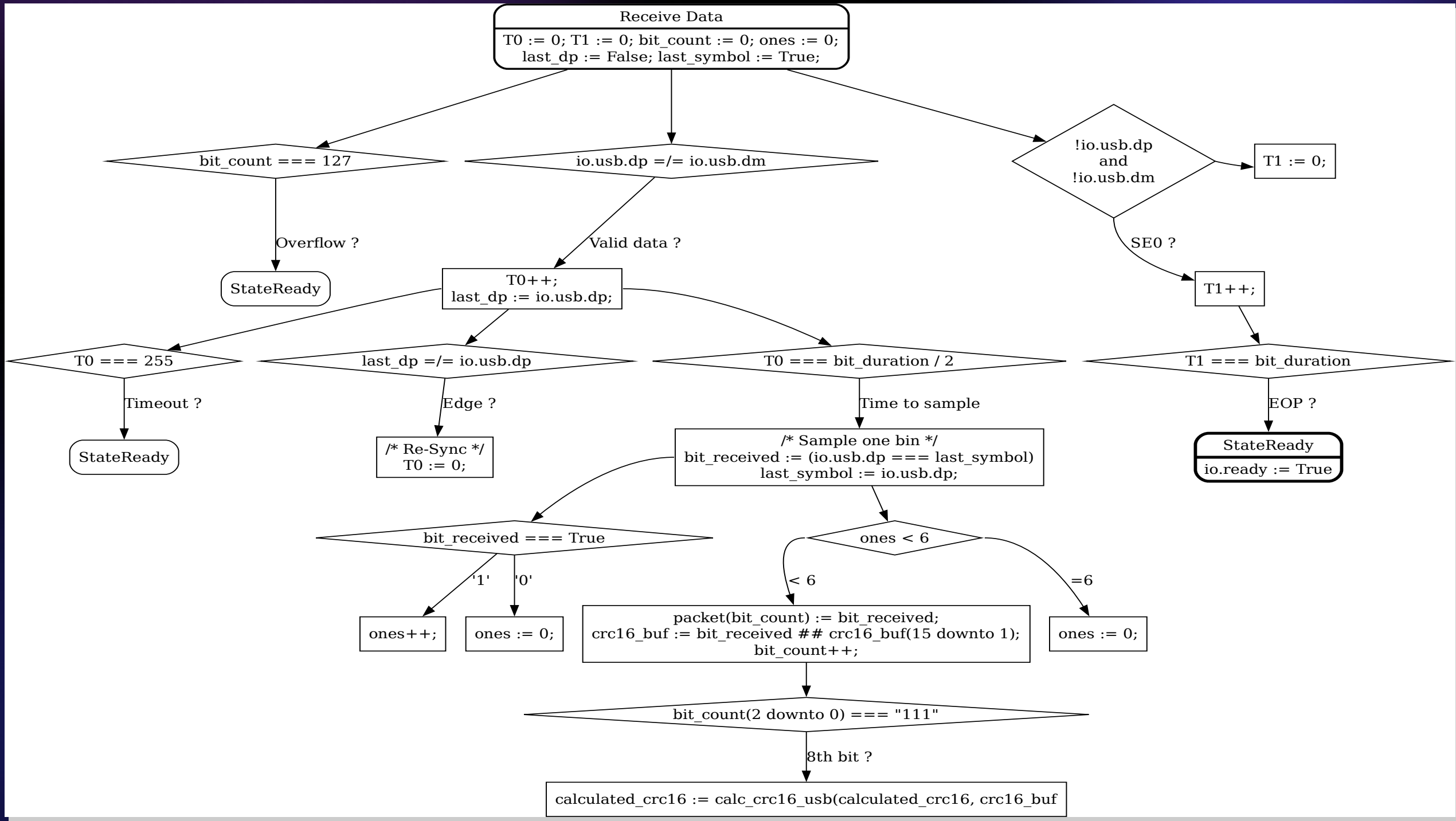
FPGA SYSTEMS
2025



Фаза 1: калибровка длительности битового интервала.



Фаза 2: ожидания конца преамбулы (SYNC).


Receive Data
T0 := 0; T1 := 0; bit_count := 0; ones := 0;
last_dp := False; last_symbol := True;
bit_count === 127
Overflow ?
StateReady
io.usb.dp =/= io.usb.dm
Valid data ?
T0++;
last_dp := io.usb.dp;
T0 === 255
Timeout ?
StateReady
last_dp =/= io.usb.dp
Edge ?
/* Re-Sync */
T0 := 0;
T0 === bit_duration / 2
Time to sample
/* Sample one bin */
bit_received := (io.usb.dp === last_symbol)
last_symbol := io.usb.dp;
bit_received === True
'1'
'0'
ones++;
ones := 0;
ones < 6
< 6
=6
packet(bit_count) := bit_received;
crc16_buf := bit_received ## crc16_buf(15 downto 1);
bit_count++;
ones := 0;
bit_count(2 downto 0) === "111"
8th bit ?
calculated_crc16 := calc_crc16_usb(calculated_crc16, crc16_buf
!io.usb.dp
and
!io.usb.dm
T1 := 0;
SE0 ?
T1++;
T1 === bit_duration
EOP ?
StateReady
io.ready := True

17a. Реализация КА хост-контроллера на SpinalHDL: базовый класс USBSendReceive

FPGA SYSTEMS
2025

При реализации вспомогательных КА применен ООП подход:

- Все вспомогательные КА имеют общий (пересекающиеся) функционал, он был собран и вынесен в базовый класс **USBSendReceive** в виде функций и процедур.

- В базовый класс вынесены общие регистры и сигналы.

- Конструктор базового класса введен ряд параметров которые использованы для условной генерации кода.

Это необходимо для удаления незадействованной аппаратуры на стадии генерации Verilog файла.

```
class USBSendReceive(
        var hasStrobe  : Boolean = true,     // если необходима генерация строба;
        var hasSendKJ  : Boolean = true,     // если КА передает «символы»
        var szBitcount : Int = 0,            // задает размерность счетчика bit_count, если 0 – нет
кода
        var hasCRC16   : Boolean = false     // если КА ведет расчет CRC16
) extends Component {

        val clock_div    = (hasStrobe   ) generate Reg(UInt(8 bits)).addTag(crossClockDomain)
        val clock_strobe = (hasStrobe   ) generate False
        val bit_count    = (szBitcount>0) generate Reg(UInt(szBitcount
bits)).addTag(crossClockDomain)
        val stuffing     = (hasSendKJ   ) generate False
        val ones         = (hasSendKJ   ) generate Reg(UInt(3 bits)).addTag(crossClockDomain)
        val last_kj      = (hasSendKJ   ) generate Reg(Bool()).addTag(crossClockDomain)
        val crc16        = (hasCRC16    ) generate Reg(Bits(16 bits)).addTag(crossClockDomain) {

        sendKJ(io: USB_IO, input_bit: Bool) = hasSendKJ generate {
                // передача одного «символа»
        }

        make_clock_strobe(io: USB_IO) = hasStrobe generate {
                // формирование строба для подсчета битовых интервалов
        }

        inc_bit_count(io: USB_IO) = (szBitcount > 0) generate {
                // приращение счетчика битовых интревалов
        }

        make_stuffing(io: USB_IO, bit_to_send: Bool) = hasSendKJ generate {
                // вычисление необходимости выполнить бит-стаффинг
        }

        calc_crc16_usb(crc_in: Bits, din: Bits) : Bits = hasCRC16 generate {
                // расчет CRC16 для 8 бит
        }

        ...
}
```

// 17b. Реализация КА USBSendSE0

FPGA SYSTEMS
2025

- Передает в конструктор базового класса параметры для условной генерации кода:
  - **hasSendKJ = false**
  - **hasStrobe = true**
  - **szBitcount = 16**

- Расширяет интерфейсный класс **USB_IO** дополнительным сигналом **len**.

- Вызывает процедуры базового класса:
  - **make_clock_strobe()**
  - **inc_bit_count()**

- Добавляет к базовому классу код реализующий основную функцию данного КА.

- Использует процедуры **sendJ()** и **sendSE0()** базового класса.

```
class USBSendSE0() extends USBSendReceive(hasSendKJ = false,
                                          hasStrobe = true,
                                          szBitcount = 16) {
    val io = new USB_IO {
        val len     = in UInt(16 bits) // number of bit intervals
    }

    make_clock_strobe(io)
    inc_bit_count(io)

    io.test := clock_strobe

    when(io.valid) {

      when(clock_strobe && bit_count === io.len) {
         // Ready, EOP: 'J'
        sendJ(io)
        io.ready := True
      } otherwise {
        sendSE0(io)
      }
    }
}
```

Порожденный код на Verilog →

// 17c. Реализация КА USBSendSE0 (порождаемый Verilog)

FPGA SYSTEMS 2025

```
module USBSendSE0 (
  inout  wire          io_usb_dm,
  inout  wire          io_usb_dp,
  input  wire          io_valid,
  output reg           io_ready,
  input  wire [7:0]    io_clock_div,
  output wire          io_test,
  input  wire [15:0]   io_len,
  input  wire          io_usb_clk,
  input  wire          bus_enable
);

  reg                 _zz_io_usb_dp;
  reg                 _zz_io_usb_dp_1;
  reg                 _zz_io_usb_dm;
  reg                 _zz_io_usb_dm_1;
  (* async_reg = "true" *) reg        [7:0]    clock_div;
  reg                 clock_strobe;
  (* async_reg = "true" *) reg        [15:0]   bit_count;
  wire                when_Apb3USB10_l112;
  wire                when_Apb3USB10_l115;
  wire                when_Apb3USB10_l350;

  assign io_usb_dm = _zz_io_usb_dm_1 ? 1'b1 : 1'bz;
  assign io_usb_dm = _zz_io_usb_dm ? 1'b0 : 1'bz;
  assign io_usb_dp = _zz_io_usb_dp_1 ? 1'b0 : 1'bz;
  assign io_usb_dp = _zz_io_usb_dp ? 1'b0 : 1'bz;
  always @(*) begin
    _zz_io_usb_dp = 1'b0;
    if(io_valid) begin
      if(!when_Apb3USB10_l350) begin
        _zz_io_usb_dp = 1'b1;
      end
    end
  end
```

```
  always @(*) begin
    _zz_io_usb_dp_1 = 1'b0;
    if(io_valid) begin
      if(when_Apb3USB10_l350) begin
        _zz_io_usb_dp_1 = 1'b1;
      end
    end
  end

  always @(*) begin
    _zz_io_usb_dm = 1'b0;
    if(io_valid) begin
      if(!when_Apb3USB10_l350) begin
        _zz_io_usb_dm = 1'b1;
      end
    end
  end

  always @(*) begin
    _zz_io_usb_dm_1 = 1'b0;
    if(io_valid) begin
      if(when_Apb3USB10_l350) begin
        _zz_io_usb_dm_1 = 1'b1;
      end
    end
  end

  always @(*) begin
    clock_strobe = 1'b0;
    if(io_valid) begin
      if(when_Apb3USB10_l115) begin
        clock_strobe = 1'b1;
      end
    end
  end
```

```
  always @(*) begin
    io_ready = 1'b0;
    if(io_valid) begin
      if(when_Apb3USB10_l350) begin
        io_ready = 1'b1;
      end
    end
  end  assign when_Apb3USB10_l112 =
        (clock_div == io_clock_div);
  assign when_Apb3USB10_l115 = (clock_div == 8'h0);
  assign io_test = clock_strobe;
  assign when_Apb3USB10_l350 = (clock_strobe &&
        (bit_count == io_len));
  always @(posedge io_usb_clk) begin
    if(io_valid) begin
      clock_div <= (clock_div + 8'h01);
      if(when_Apb3USB10_l112) begin
        clock_div <= 8'h0;
      end
    end else begin
      clock_div <= 8'h0;
    end
    if(io_valid) begin
      if(clock_strobe) begin
        bit_count <= (bit_count + 16'h0001);
      end
    end else begin
      bit_count <= 16'h0;
    end
  end

endmodule
```

```
$ wc USBSendSE0.sv
  108   260  2553 USBSendSE0.sv
```

// 17d. Реализация КА USBSendShortToken

FPGA SYSTEMS
2025

- Передает в конструктор базового класса параметры для условной генерации кода:
  szBitcount = 5
остальные — принимаю значения по умолчанию.

- Расширяет интерфейсный класс **USB_IO** дополнительным сигналом **pid**.

- Расширяет базовый класс USBSendReceive дополнительными сигналами **buffer** и **bit_to_send**.

- Вызывает процедуры базового класса:
  make_clock_strobe()
  inc_bit_count()
  reset_last_kj()

- Добавляет к базовому классу код реализующий основную функцию данного КА используя:
sendJ(), **sendKJ()**, и **sendSE0()**.

```
class USBSendShortToken() extends USBSendReceive(szBitcount = 5) {
    val io = new USB_IO {
        val pid       = in Bits(4 bits) // pid to send
    }

    val buffer = ~io.pid ## io.pid ## B"10000000"
    val bit_to_send = buffer(bit_count(3 downto 0))

    make_clock_strobe(io)
    inc_bit_count(io)
    reset_last_kj(io)

    io.test := clock_strobe

    when(io.valid) {
      when(bit_count === 20) { // Ready, EOP: 'J'
        sendJ(io)
        io.ready := True
        bit_count := 20
      } elsewhen(bit_count === 19) { // EOP: 'J'
        sendJ(io)
      } elsewhen(bit_count === 18 && clock_strobe) { // EOP: 'SE0' - corner case
        sendSE0(io)
      } elsewhen((bit_count === 17) || (bit_count === 18)) { // EOP: 'SE0'
        sendSE0(io)
      } elsewhen(bit_count === 16 && clock_strobe) { // EOP: 'SE0' - corner case
        sendSE0(io)
      } otherwise {
        sendKJ(io, bit_to_send)
      }
    }
}
```

// 17e. Реализация КА USBSendLongToken

FPGA SYSTEMS
2025

- Передает в конструктор базового класса параметры для условной генерации кода:
  szBitcount = 6
остальные — принимаю значения по умолчанию.

- Расширяет интерфейсный класс **USB_IO** дополнительными сигналами **pid, addr** и **endp**.

- Расширяет базовый класс USBSendReceive дополнительной функцией **calc_crc5_usb()** которая возвращает CRC5 код.

- Расширяет базовый класс USBSendReceive дополнительными сигналами **buffer**, **bit_to_send и crc5_out**.

- Вызывает процедуры базового класса:
  make_clock_strobe()
  inc_bit_count()
  reset_last_kj()

- Добавляет к базовому классу код реализующий основную функцию данного КА используя:
sendJ(), **sendKJ()**, и **sendSE0()**.

```
class USBSendLongToken() extends USBSendReceive(szBitcount = 6) {
    val io = new USB_IO {
        val pid       = in Bits(4 bits)
        val addr      = in Bits(7 bits)
        val endp      = in Bits(4 bits)
    }

    def calc_crc5_usb(din: Bits) : Bits = {
      val ret = Bits(5 bits)
      val din_rev = din.reversed;
      ret(0) := din_rev(10) ^ din_rev(9) ^ din_rev(6) ^ din_rev(5) ^ din_rev(3) ^ din_rev(0) ^ True
      ret(1) := din_rev(10) ^ din_rev(7) ^ din_rev(6) ^ din_rev(4) ^ din_rev(1) ^ True
      ret(2) := din_rev(10) ^ din_rev(9) ^ din_rev(8) ^ din_rev(7) ^ din_rev(6) ^ din_rev(3) ^ din_rev(2)
^ din_rev(0) ^ True
      ret(3) := din_rev(10) ^ din_rev(9) ^ din_rev(8) ^ din_rev(7) ^ din_rev(4) ^ din_rev(3) ^ din_rev(1)
      ret(4) := din_rev(10) ^ din_rev(9) ^ din_rev(8) ^ din_rev(5) ^ din_rev(4) ^ din_rev(2) ^ True
      return ret.reversed ^ B"11111"
    }

    val buffer = crc5_out ## io.endp ## io.addr ## ~io.pid ## io.pid ## B"10000000"
    val bit_to_send = buffer(bit_count(4 downto 0))
    val crc5_out = calc_crc5_usb(io.endp ## io.addr)

    make_clock_strobe(io)
    inc_bit_count(io)
    reset_last_kj(io)

    when(io.valid) {
      when(bit_count === 36) { // Ready, EOP: 'J'
        sendJ(io)
        io.ready := True
        bit_count := 36
      } elsewhen(bit_count === 35) { // EOP: 'J'
        sendJ(io)
      } elsewhen((bit_count === 33) || (bit_count === 34)) { // EOP: 'SE0'
        sendSE0(io)
      } elsewhen(bit_count === 32 && clock_strobe) { // EOP: 'SE0' - corner case
        sendSE0(io)
      } otherwise {
        sendKJ(io, bit_to_send)
      }
    }
}
```

// 17f. Реализация КА USBSendData

FPGA SYSTEMS
2025

- Передает в конструктор базового класса параметры для условной генерации кода:
  hasCRC16 = true,
  szBitcount = 8
остальные — принимаю значения по умолчанию.

- Расширяет интерфейсный класс **USB_IO** дополнительными сигналами **pid, data** и **len**.

- Расширяет базовый класс USBSendReceive дополнительными регистрами: **state**, **buffer**,
и сигналами: **bit_to_send** и **crc_byte**.

- Вызывает процедуры базового класса:
  make_clock_strobe()
  make_stuffing()
  reset_last_kj()

- Добавляет к базовому классу код реализующий основную функцию данного КА используя:
sendJ(), **sendKJ()**, и **sendSE0()**.

- Использует функцию **calc_crc16_usb()** базового класса для вычисления кода CRC16.

```
class USBSendData() extends USBSendReceive(hasCRC16 = true, szBitcount = 8) {
    val io = new USB_IO {
        val pid       = in Bits(4 bits)
        val data      = in Bits(64 bits)
        val len       = in UInt(7 bits)
    }

    val state = Reg(UInt(3 bits)).addTag(crossClockDomain) init(0)
    val crc_byte = Reg(Bits(8 bits))
    val buffer = ~io.pid ## io.pid ## B"10000000"
    val bit_to_send = False // Value of data bit to be sent

    make_clock_strobe(io)
    make_stuffing(io, bit_to_send)
    reset_last_kj(io)

    when(io.valid) {
      switch(state) {
        is(0) { // sending SYNC + PID
          bit_to_send := buffer(bit_count(3 downto 0))
          sendKJ(io, bit_to_send)
          when(clock_strobe && bit_count === 15) {
            bit_count := 0
            state := 1 // send data
            when(io.len === 0x7f) { state := 2 } // max len means send empty packet, go and send CRC16 right away
          }
        }
        is(1) { // sending DATA block
          bit_to_send := io.data(bit_count(5 downto 0))
          when(clock_strobe && !stuffing) {
            val crc_byte_next = bit_to_send ## crc_byte(7 downto 1)
            when(bit_count(2 downto 0) === U"111") {
              crc16 := calc_crc16_usb(crc16, crc_byte_next.reversed)
            }
            when(bit_count === io.len) {
              state := 2
              bit_count := 0
            }
            crc_byte := crc_byte_next
          }
          sendKJ(io, bit_to_send) // KJKJKJKK + PID + DATA + CRC16
        }
        is(2) { // sending CRC16 block
          val crc_rev = ~crc16.reversed // CRC out is reversed and XORed
          bit_to_send := crc_rev(bit_count(3 downto 0))
          when(clock_strobe && bit_count === 16) {
            sendSE0(io)
            state := 3
          } otherwise {
            sendKJ(io, bit_to_send)
          }
        }
        is(3) { // sending EOP: 'SE0'
          sendSE0(io)
          when(clock_strobe) { state := 4 }
        }
        is(4) { // sending EOP: 'SE0'
          sendSE0(io)
          when(clock_strobe) { // sending EOP: 'J'
            sendJ(io)
            state := 5
          }
        }
        is(5) { // sending EOP: 'J'
          sendJ(io)
          when(clock_strobe) { state := 6 }
        }
        is(6) { // Ready, EOP: 'J'
          sendJ(io)
          io.ready := True
        }
      }
    } otherwise {
      state := 0
      crc16 := B"16'hFFFF"
    }
}
```

// 17g. Реализация КА USBRecevier

FPGA SYSTEMS
2025

- Передает в конструктор базового класса параметры для условной генерации кода:
  - **hasStrobe = false,**
  - **hasSendKJ = false,**
  - **hasCRC16 = true,**
  - **szBitcount = 7**

- Расширяет интерфейсный класс **USB_IO** дополнительными сигналами **packet, bits_recv, calculated_crc16** и **received_crc16**.

- Расширяет базовый класс USBSendReceive дополнительными регистрами: **state, ones, bit_duration, received_crc16, last_symbol, last_dp, packet, и т.д.**

- Добавляет к базовому классу код реализующий основную функцию данного КА.

- Не использует строб, самостоятельно реализует приращение **bit_count** в зависимости от фазы.

- Использует функцию **calc_crc16_usb()** базового класса для вычисления кода CRC16.

```
class USBReceiver() extends USBSendReceive(hasStrobe = false, hasSendKJ = false, hasCRC16 =
true, szBitcount = 7) {
    val io = new USB_IO {
        val packet    = out Bits(128 bits) // PID(8) + DATA(64) + CRC16(16) + ALIGN
        val bits_recv = out UInt(7 bits)
        val calculated_crc16 = out Bits(16 bits)
        val received_crc16 = out Bits(16 bits)
    }

    override val ones = Reg(UInt(3 bits)).addTag(crossClockDomain)
    val received_crc16 = Reg(Bits(16 bits)).addTag(crossClockDomain) init(0)
    val bit_duration = Reg(UInt(8 bits)).addTag(crossClockDomain) init(0)
    val state = Reg(UInt(3 bits)).addTag(crossClockDomain) init(0)
    val T0 = Reg(UInt(8 bits)).addTag(crossClockDomain) init(0) // Bit timer
    val T1 = Reg(UInt(8 bits)) init(0) // EOP timer
    val T2 = Reg(UInt(12 bits)) init(0) // Guard timer

    val ready = Reg(Bool()).addTag(crossClockDomain) init(False)
    val packet = Reg(Bits(128 bits)).addTag(crossClockDomain) init(0)
    val last_dp = Reg(Bool()) init(True)
    val last_symbol = Reg(Bool()) init(True)

    io.packet := packet
    io.bits_recv := bit_count
    io.calculated_crc16 := ~crc16.reversed
    io.received_crc16 := received_crc16

    when(io.valid) {
      io.ready := ready

      switch(state) {
        is(0) { // SYNC: begin calibration
          when(io.usb.dp && !io.usb.dm) { // First 'K' - start calibration
            state := 1
            bit_count := 0
            ones := 0
            bit_duration := 7 // default is 8 clocks
            packet := 0
            crc16 := B"16'hFFFF"
            ready := False
            last_dp := True
            last_symbol := True
            T0 := 1 // compensation for init state which takes just one clock
            T1 := 0
            T2 := 0
          }
        }
        is(1) { // SYNC: 'K' is going, waiting for 'J'
          T0 := T0 + 1
          when(!io.usb.dp && io.usb.dm) { // 'J' received
            bit_count := bit_count + 1
            state := 2
          }
          when(T0 === 255) { // Too much, error
            state := 7
          }
        }
        is(2) { // SYNC: 'J' is going, waiting for 'K'
          T0 := T0 + 1
          when(io.usb.dp && !io.usb.dm) { // 'K' received
            bit_count := bit_count + 1
            state := 1
            when(bit_count === 3) { // TWO 'KJ' cycles received
              bit_duration := (T0 >> 2).resized // calculate bit duration: div by 4
              state := 3
              T0 := 0
            }
          }
          when(T0 === 255) { // Too much, error
            state := 7
          }
        }
        is(3) { // Wait for end of SYNC: two 'K's
          when(io.usb.dp && !io.usb.dm) { // 'K' received
            T0 := T0 + 1
            when(T0.asBits === bit_duration(6 downto 0) ## B"0") {
                                          // T0 === bit_duration * 2
              T0 := 0
              state := 4
              bit_count := 0
            }
          } otherwise {
            T0 := 0
          }
          when(T0 === 255) { // Too much, error
            state := 7
          }
        }
```

```
        is(4) { // Receiving data
          when(io.usb.dp =/= io.usb.dm) { // Valid data are only when DP != DM
            T0 := T0 + 1
            last_dp := io.usb.dp
            when(last_dp =/= io.usb.dp) { // sync on each edge
              T0 := 0
            }
            when(T0 === bit_duration) { // end of symbol ?
              T0 := 0
            }
            when(T0 === bit_duration(7 downto 1).resized) { // sample one symbol in middle of
tick
              var bit_received = (io.usb.dp === last_symbol) // convert symbol to bit
              last_symbol := io.usb.dp
              when(bit_received) {
                ones := ones + 1
              } otherwise {
                ones := 0
              }
              when(ones =/= 6) { // save current bit
                bit_count := bit_count + 1
                packet(bit_count) := bit_received
                received_crc16 := bit_received ## received_crc16(15 downto 1)
                when(bit_count > 23 && bit_count(2 downto 0) === U("000")) {
                        crc16 := calc_crc16_usb(crc16, received_crc16(7 downto 0).reversed)
                }
              } otherwise { // skip current bit because it's stuffing bit
                ones := 0
              }
              when(bit_count === 127) { // max data size achieved
                state := 7
              }
              io.test := True
            }
          }
        }
        is(6) { // EOP received
          T0 := T0 + 1
          when(T0 === bit_duration) { // wait for 'J' after SE0
            state := 7
          }
        }
        is(7) { // Ready
          ready := True
          io.bits_recv := bit_count
          io.test := True
        }

        default {
          state := 7
          packet(87 downto 80) := state.asBits.resized
          io.test := True
        }

      }

      // Check for EOP (SE0)
      when(!io.usb.dp && !io.usb.dm) {
        T1 := T1 + 1
        when(T1 === ((bit_duration << 1) - U(2))) { // is SE0 for two bit intervals - report EOP
          state := 6
          T0 := 0
        }
      } otherwise {
        T1 := 0
      }

      // Check for hung state ('J')
      when(!io.usb.dp && io.usb.dm) {
        T2 := T2 + 1
        when(T2 === (bit_duration << 3)) { // is 'J' for more than 8 clocks - report error!
          state := 7
        }
      } otherwise {
        T2 := 0
      }

    } otherwise {
      state := 0
      ready := False
    }
}
```

// 18a. Реализация основного КА: класс Apb3USB10Ctrl

FPGA SYSTEMS
2025

Реализация основного КА находится в классе **Apb3USB10Ctrl**, который помимо КА содержит:

- Объявление внешних интерфейсных сигналов: **apb**, **usb**, **interrupt** и **usb_clk**.

- Привязку к шине ABP3 в качестве slave устройства с помощью библиотечного класса **Apb3SlaveFactory**.

- Создание программно доступных регистров управления в общей памяти с помощью функций **createReadOnly()** и **createReadWrite()** класса **Apb3SlaveFactory**.

- Разбиение регистров на битовые поля с поименованными сигналами.

- Описание тактового домена для USB хост-контроллера с помощью класса **ClockDomain**.

- Создание области кода с помощью класса **ClockingArea** которая будет тактироваться и управляться сигналами в этом тактовом домена.

Реализация всех КА USB хост-контроллера находится в этой области.

```
class Apb3USB10Ctrl(usbFrequency : HertzNumber = 12.0 MHz) extends Component {
  val io = new Bundle {
    val apb       = slave(Apb3(addressWidth = 12, dataWidth = 32))
    val usb       = master(USBInterface())
    val interrupt = out Bool()
    val usb_clk   = in Bool()

    val test = out Bool()
  }

  val busCtrl = Apb3SlaveFactory(io.apb)

  // 0x00 - STATUS
  val usbStatusWord = busCtrl.createReadOnly(Bits(32 bits), address = 0) init(0)
  val error_flag = usbStatusWord(30).addTag(crossClockDomain)
  val report_flag = usbStatusWord(29).addTag(crossClockDomain)
  val busy_flag = usbStatusWord(28).addTag(crossClockDomain)
  val received_flag = usbStatusWord(27).addTag(crossClockDomain)
  val crc16_ok_flag = usbStatusWord(26).addTag(crossClockDomain)
  // ... more flags here
  //val received_pid = usbStatusWord(23 downto 16).addTag(crossClockDomain)
  // ... reserved for FSM states
  val fsm_state = usbStatusWord(2 downto 0).addTag(crossClockDomain)

  // 0x04 - COMMAND
  val usbCommandWord = busCtrl.createReadWrite(Bits(32 bits), address = 4) init(0)
  val cmd_start = usbCommandWord(31).addTag(crossClockDomain)
  val cmd_addr = usbCommandWord(30 downto 24).addTag(crossClockDomain)
  val cmd_endp = usbCommandWord(23 downto 20).addTag(crossClockDomain)
  val cmd_len = usbCommandWord(14 downto 8).asUInt.addTag(crossClockDomain)
  val cmd_pid = usbCommandWord(7 downto 4).addTag(crossClockDomain)
  val cmd = usbCommandWord(3 downto 0).addTag(crossClockDomain
  ...

  val usbClockDomain = ClockDomain(
    clock = io.usb_clk,
    reset = bus_enable,
    config = ClockDomainConfig(resetKind = SYNC, resetActiveLevel = LOW),
    frequency = FixedFrequency(usbFrequency)
  )

  val usb_area = new ClockingArea(usbClockDomain) {

      // Реализация основного КА находится здесь

  }
}
```

18b. Реализация основного КА: вычисление параметров и объявление регистров

FPGA SYSTEMS 2025

- Вычисляются три многобитовых сигнала для параметров КА:
  - **USBLowSpeedClockDiv** — для получения 1,5 МГц;
  - **USBLowSpeedKeepAliveClocks** — интервал для «KeepAlive»;
  - **USBLowSpeedErrorClocks** — время детектирования состояния «disconnect».

- Объявляются регистр состояния: **state**, регистры счетчиков для таймеров:
  - **T1** — защитный таймер;
  - **T2** — время до следующего «KeepAlive».

- Объявляются внутренние регистры флагов:
  - **error** — обнаружена ошибка на шине;
  - **report** — источник событий для контроллера прерываний;
  - **busy** — хост-контроллер занят;
  - **received** — принят пакет;
  - **crc16_ok** — у принятого пакета CRC16 совпал.

- Объявляются и формируются вспомогательные сигналы:
  - **bus_error** — шина пребывает в аварийном состоянии;
  - **bus_present** — обнаружено подключение устройства («J»);
  - **bus_activity** — устройство пытается что-то послать («K»);

- Внутренние флаги транслируются в программно-доступные.

```
  val usb_area = new ClockingArea(usbClockDomain) {

    println("Apb3USB10Ctrl::usbFrequency = %d Hz".format(usbFrequency.toInt));

    val USBLowSpeedClockDiv = UInt(8 bits)
    val low_speed_baudrate : HertzNumber = 1.5 MHz;
    val USBLowspeedclockdiv = (ClockDomain.current.frequency.getValue / low_speed_baudrate + 0.5).toBigInt
- 1
    USBLowSpeedClockDiv := USBLowspeedclockdiv
    println("Apb3USB10Ctrl::USBLowSpeedClockDiv = %d".format(USBLowspeedclockdiv));

    val USBLowSpeedKeepAliveClocks = UInt(16 bits)
    val low_speed_keepalive : TimeNumber = 1.0 ms; // Send KeepAlive interval
    val usblowspeedkeepaliveclocks = (ClockDomain.current.frequency.getValue * low_speed_keepalive +
0.5).toBigInt - 1
    USBLowSpeedKeepAliveClocks := usblowspeedkeepaliveclocks
    println("Apb3USB10Ctrl::USBLowSpeedKeepAliveClocks = %d".format(usblowspeedkeepaliveclocks));

    val USBLowSpeedErrorClocks = UInt(16 bits)
    val low_speed_error : TimeNumber = 0.8 ms; // Time to detect disconnect or error
    val usblowspeederrorclocks = (ClockDomain.current.frequency.getValue * low_speed_error + 0.5).toBigInt
- 1
    USBLowSpeedErrorClocks := usblowspeederrorclocks
    println("Apb3USB10Ctrl::USBLowSpeedErrorClocks = %d".format(usblowspeederrorclocks));

    val state = RegInit(StateUnconnected).addTag(crossClockDomain)
    val T1 = Reg(UInt(16 bits)).addTag(crossClockDomain) init(0) // Guard timer
    val T2 = Reg(UInt(16 bits)).addTag(crossClockDomain) init(0) // Low-Speed Keep-Alive timer

    val error = Reg(Bool()).addTag(crossClockDomain) init(False)
    val report = Reg(Bool()).addTag(crossClockDomain) init(False)
    val busy = Reg(Bool()).addTag(crossClockDomain) init(False)
    val received = Reg(Bool()).addTag(crossClockDomain) init(False)
    val crc16_ok = Reg(Bool()).addTag(crossClockDomain) init(False)

    val bus_error = !io.usb.dm && !io.usb.dp; // Both DP and DM low means nothing is connected
    val bus_present = io.usb.dm && !io.usb.dp; // Low Speed device connected
    val bus_activity = !io.usb.dm && io.usb.dp; // Polarity change 'K' indicates some activity

    error_flag := error
    report_flag := report
    received_flag := received
    busy_flag := busy
    crc16_ok_flag := crc16_ok
    fsm_state := state.asBits
```

18c. Подключение вспомогательных КА

FPGA SYSTEMS
2025

- Подключаются (инстанциируются) вспомогательные КА:
  переменная **send_se0** для КА **USBSendSE0**;
  переменная **send_short_token** для КА **USBSendShortToken**;
  переменная **send_long_token** для КА **USBSendLongToken**;
  переменная **send_data** для **USBSendData**;
  переменная **receiver** для КА **USBReceiver**;

- Всем КА устанавливаются входные сигналы по умолчанию.

- Всем КА передается длительность битового интервала в входном сигнале **io.clock_div**;

- Все КА по умолчанию не активны (**io.valid := False**).

- У КА **USBReceiver** нет входных сигналов кроме **io.valid**.

```
val send_se0 = new USBSendSE0()
send_se0.io.valid := False
send_se0.io.len := 0
send_se0.io.clock_div := USBLowSpeedClockDiv

val send_short_token = new USBSendShortToken()
send_short_token.io.pid := 0
send_short_token.io.valid := False
send_short_token.io.clock_div := USBLowSpeedClockDiv

val send_long_token = new USBSendLongToken()
send_long_token.io.pid := 0
send_long_token.io.addr := 0
send_long_token.io.endp := 0
send_long_token.io.valid := False
send_long_token.io.clock_div := USBLowSpeedClockDiv

val send_data = new USBSendData()
send_data.io.pid := 0
send_data.io.data := 0
send_data.io.len := 0x7f // zero data bits
send_data.io.valid := False
send_data.io.clock_div := USBLowSpeedClockDiv

val receiver = new USBReceiver()
receiver.io.valid := False
```

// 18d. Обработка команд и состояний основного КА

FPGA SYSTEMS
2025

- Если установлен флаг **bus_present**, то КА переходит в состояние **StateWaitCMDorSYNC** в котором он может:
  - принимать команды от ПО;
  - активировать автомат приема данных;
  - активировать автомат генерации «KeepAlive»

- Команды интерпретируются если установлен бит **cmd_start** программно доступного регистра **usbCommandWord**;

- При исполнении команды автомат переходит в соответствующее состояния и остается в нем до её завершения;

- Если установлен флаг **bus_activity** и КА находится в состоянии **StateWaitCMDorSYNC**, то активируется автомат приема пакета **USBReceiver**.

```
object USBMain extends SpinalEnum {
  val StateUnconnected, StateWaitCMDorSYNC, StateKeepAlive, StateSendLongToken,
        StateSendShortToken, StateSendData, StateSendReset, StateReceive          = newElement()
}

object USBCommand extends SpinalEnum(defaultEncoding = binarySequential) {
  val CMDNone, CMDSendLongToken, CMDSendShortToken, CMDSendData, CMDBusReset          = newElement()
}

switch(state) {

  is(StateUnconnected) { // Unconnected
    report := False

    when(bus_present) {
      busy := False
      cmd_start := False
      error := False
      report := True
      state := StateWaitCMDorSYNC // Low Speed device just connected
    }
  }

  is(StateWaitCMDorSYNC) { // Wait command or SYNC
    report := False

    when(cmd_start) {
      switch(cmd) {
        is(CMDSendLongToken.asBits.resize(4)) {
          state := StateSendLongToken
          busy := True
          received := False
        }
        is(CMDSendShortToken.asBits.resize(4)) {
          state := StateSendShortToken
          busy := True
          received := False
        }
        is(CMDSendData.asBits.resize(4)) {
          state := StateSendData
          busy := True
          received := False
        }
        is(CMDBusReset.asBits.resize(4)) {
          state := StateSendReset
          busy := True
          received := False
        }
        default {
          cmd_start := False
          report := True
          received := False
        }
      }
    }

    when(bus_activity) { // Activity on the bus ?
      state := StateReceive
      busy := True
      received := False
    }

    when(keepalive_enable && !(bus_error)) { // Keepalive enabled and not Error state ?
      T2 := T2 + 1
      when(T2 === USBLowSpeedKeepAliveClocks) {
        state := StateKeepAlive
        busy := True
      }
    }
  }
```

// 18e. Активация вспомогательных КА

FPGA SYSTEMS
2025

При активации вспомогательного КА:

- На входные сигналы копируются данные из программно доступных регистров;
- Производится коммутация шины USB:
  (**send_data.io.usb <> io.usb**);
- Устанавливается флаг **busy**;
- Устанавливается соответствующий сигнал **io.valid**;

В процессе исполнения команды основной КА ожидает установки сигнала готовности **io.ready** от соответствующего КА.

По готовности снимается сигнал **io.valid**, а также флаги **busy** и **cmd_start**, поднимается флаг **report**. Обнуляется защитный таймер **T1**.

```
      is(StateSendLongToken) { // Send Long Token
        send_long_token.io.pid := cmd_pid
        send_long_token.io.addr := cmd_addr
        send_long_token.io.endp := cmd_endp
        send_long_token.io.valid := True
        send_long_token.io.usb <> io.usb
        when(send_long_token.io.ready) {
          state := StateWaitCMDorSYNC
          report := True
          cmd_start := False
          busy := False
          T1 := 0
        }
      }

      is(StateSendShortToken) { // Send Short Token
        send_short_token.io.pid := cmd_pid
        send_short_token.io.valid := True
        send_short_token.io.usb <> io.usb
        when(send_short_token.io.ready) {
          state := StateWaitCMDorSYNC
          report := True
          cmd_start := False
          busy := False
          T1 := 0
        }
      }

      is(StateSendData) { // send DATA packet
        send_data.io.pid := cmd_pid
        send_data.io.data := send_data_high ## send_data_low
        send_data.io.len := cmd_len // in bits - 1
        send_data.io.valid := True
        send_data.io.usb <> io.usb
        when(send_data.io.ready) {
          state := StateWaitCMDorSYNC
          report := True
          cmd_start := False
          busy := False
          T1 := 0
        }
      }

      is(StateSendReset) { // Bus Reset condition is SE0 (D+ and D- are low) for 11ms
          // ...
      }

      is(StateKeepAlive) { // KeepAlive (Low-Speed only) is SE0 for just two bit intervals
          // ...
      }

      is(StateReceive) { // Receive data piece
        receiver.io.valid := True
        receiver.io.usb <> io.usb
        when(receiver.io.ready) {
          state := StateWaitCMDorSYNC
          received := True
          busy := False
          report := True
          T1 := 0
          received_pid := receiver.io.packet(7 downto 0)
          received_data_low := receiver.io.packet(39 downto 8)
          received_data_high := receiver.io.packet(71 downto 40)
          received_bits := receiver.io.bits_recv.asBits.resized
          received_crc16 := receiver.io.received_crc16 //receiver.io.packet(87 downto 72)
          calculated_crc16 := receiver.io.calculated_crc16
          crc16_ok := receiver.io.received_crc16 === receiver.io.calculated_crc16
        }
      }
    } // switch(state)
  } // usb_area
}
```

// 19. Интеграция USB хост-контроллера в СнК «KarnixSoC»

FPGA SYSTEMS
2025

1. Класс **KarnixSOC** описывает СнК как компонент. Необходимо:

- добавить шину USB в перечень внешних сигналов (в структуру **io**);
- добавить сигнал линии тактирования **usb_clk** для USB контроллера;
- инстанциировать объект класса **Apb3USB10Ctrl**;
- подключить его к шине USB;
- подключить к контроллеру прерываний (PLIC);
- подключить к мосту APB3 и назначить базовый адрес (**0xD1000**);

2. Класс **KarnixSOCTopLevel** описывает модуль уровня Top Level для всего проекта. Необходимо:

- объявить внешние вывод для шины USB в структуре **io**;
- соединить внешнюю шину USB с шиной USB внутри KarnixSOC;
- синтезировать тактовый сигнал **usb_clk** частотой ~12МГц, пропустить через блок **DCCA**;
- подключить **usb_clk** к хост-контроллеру внутри KarnixSOC;

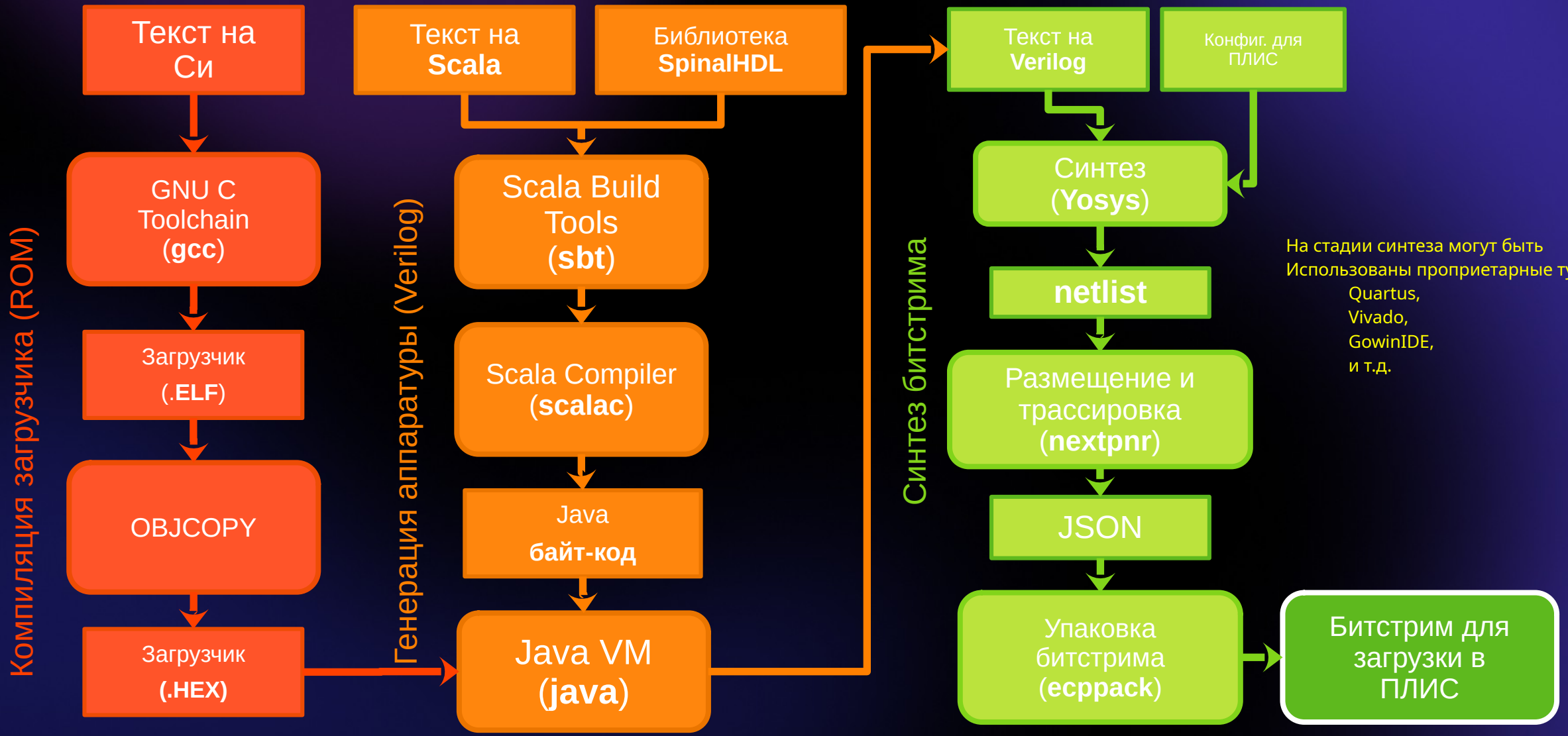
3. Добавить конфигурацию выводов микросхемы ПЛИС в файл **karnix_cabga256.lpf**:

```
LOCATE COMP "io_usb1_dm" SITE "R4";     # USB1_DM/GPIO_22
IOBUF PORT "io_usb1_dm" IO_TYPE=LVCMOS33;

LOCATE COMP "io_usb1_dp" SITE "T3";     # USB1_DP/GPIO_23
IOBUF PORT "io_usb1_dp" IO_TYPE=LVCMOS33;
```

```
class KarnixSOC(val config: KarnixSOCConfig) extends Component{
  val io = new Bundle {
    ...
    val usb1 = master(USBInterface())
    val usb_clk = in Bool()
  }
    ...
    val usb1Ctrl = new Apb3USB10Ctrl(usbFrequency) // Instantiate USB 1.0 Host Controller
    io.usb1 <> usb1Ctrl.io.usb                     // Connect to USB bus
    usb1Ctrl.io.usb_clk := io.usb_clk              // Assign clock
    plic.setIRQ(usb1Ctrl.io.interrupt, 10) // Connect to PLIC
    ...
    val apbDecoder = Apb3Decoder(
      master = apbBridge.io.apb,
      slaves = List(
        ...
        usb1Ctrl.io.apb -> (0xD1000, 4 kB) // Add USB contrller to APB3 Bridge, set base address
      )
    )
}

class KarnixSOCTopLevel() extends Component{
    val io = new Bundle {
        ...
        val usb1 = master(USBInterface()) // Define pins for USB bus
    }
    ...
    val karnix_soc = new KarnixSOC(…)      // Instantiate KarnixSOC

    karnix_soc.io.usb1 <> io.usb1          // Connect USB Host controller to external USB bus

    // Generate ~12 MHz USB1.x clock by dividing pixclk_x10 250MHz by 21
    val pixclk_x10_ClockDomain = ClockDomain(
      clock = karnix_soc.io.pixclk_x10,
      config = ClockDomainConfig(resetKind = BOOT),
      frequency = FixedFrequency(250.0 MHz)
    )

    val pixclk_x10_area = new ClockingArea(pixclk_x10_ClockDomain) {

      val usb_clk_div = Reg(UInt(5 bits)) init(0)
      val usb_clk = Reg(Bool()) init(False)

      usb_clk_div := usb_clk_div + 1
      when(usb_clk_div === 10) {
        usb_clk := True
      }
      when(usb_clk_div === 20) {
        usb_clk := False
        usb_clk_div := 0
      }

      var dcca_usb = DCCA()
      dcca_usb.CE := True
      dcca_usb.CLKI := usb_clk
      karnix_soc.io.usb_clk := dcca_usb.CLKO       // Assigned generated clock
    }
}
```

// 20a. Сборка проекта (SpinalHDL work-flow)

FPGA SYSTEMS
2025



// 20b. Сборка проекта (синтез битстрима)

FPGA SYSTEMS
2025

Сборкой всего проекта управляет **Makefile** две основные цели:

make generate — выполняет генерацию кода Verilog;

make compile — запускает синтез битстрима.

Есть возможность собрать и запустить симуляцию в Verilator: **make sim**

```
rz@devbox:$ make generate compile

(cd ../../..; sbt "runMain vexriscv.demo.KarnixSOCVerilog")
[info] welcome to sbt 1.6.0 (Ubuntu Java 11.0.9.1)
[info] loading settings for project karnixsoc-build from
plugins.sbt ...
[info] loading project definition from /home/rz/KarnixSOC/project
[info] loading settings for project root from build.sbt ...
[info] set current project to VexRiscv (in build
file:/home/rz/KarnixSOC/)
[info] compiling 1 Scala source to
/home/rz/KarnixSOC/target/scala-2.12/classes ...
[warn] four deprecations (since ???); re-run with -deprecation for
details
[warn] one warning found

[info] running (fork) vexriscv.demo.KarnixSOCVerilog
[info] [Runtime] SpinalHDL v1.10.2a    git head :
a348a60b7e8b6a455c72e1536ec3d74a2ea16935
[info] [Runtime] JVM max memory : 8294.0MiB
[info] [Runtime] Current date : 2025.10.08 21:02:51
[info] [Progress] at 0.000 : Elaborate components
[info] Apb3USB10Ctrl::usbFrequency = 11900000 Hz
[info] Apb3USB10Ctrl::USBLowSpeedClockDiv = 7
[info] Apb3USB10Ctrl::USBLowSpeedKeepAliveClocks = 11899
[info] Apb3USB10Ctrl::USBLowSpeedErrorClocks = 9519

[info] [Progress] at 6.443 : Checks and transforms
[info] [Progress] at 9.224 : Generate Verilog to .
[info] [Warning] 729 signals were pruned. You can call printPruned
on the backend report to get more informations.
[info] [Done] at 11.297
[success] Total time: 36 s, completed Oct 8, 2025, 9:03:03 PM

yosys -l yosys.log -v2 -p "synth_ecp5 -abc9 -top KarnixSOCTopLevel
-json bin/KarnixSOCTopLevel.json"
../../../KarnixSOCTopLevel.v
../../../src/main/verilog/TMDS_encoder.sv
../../../src/main/verilog/OBUFDS.sv
1. Executing Verilog-2005 frontend: ../../../KarnixSOCTopLevel.v
...

Info: Device utilisation:
Info:            TRELLIS_IO:     121/    197    61%
Info:                  DCCA:       6/     56    10%
Info:                DP16KD:      55/     56    98%
Info:            MULT18X18D:       9/     28    32%
Info:                ALU54B:       0/     14     0%
Info:               EHXPLLL:       2/      2   100%
Info:               EXTREFB:       0/      1     0%
Info:                  DCUA:       0/      1     0%
Info:             PCSCLKDIV:       0/      2     0%
Info:               IOLOGIC:       0/    128     0%
Info:              SIOLOGIC:       0/     69     0%
Info:                   GSR:       0/      1     0%
Info:                 JTAGG:       0/      1     0%
Info:                  OSCG:       0/      1     0%
Info:                 SEDGA:       0/      1     0%
Info:                   DTR:       0/      1     0%
Info:               USRMCLK:       0/      1     0%
```

```
Info: Device utilisation:
Info:                CLKDIVF:       0/      4     0%
Info:              ECLKSYNCB:       0/     10     0%
Info:                DLLDELD:       0/      8     0%
Info:                 DDRDLL:       0/      4     0%
Info:                DQSBUFM:       0/      8     0%
Info:        TRELLIS_ECLKBUF:       0/      8     0%
Info:           ECLKBRIDGECS:       0/      2     0%
Info:                   DCSC:       0/      2     0%
Info:             TRELLIS_FF:    8417/  24288    34%
Info:           TRELLIS_COMB:   22947/  24288    94%
Info:           TRELLIS_RAMW:     697/   3036    22%
...

Info: Max frequency for clock '$glbnet$io_lan_rxclk$TRELLIS_IO_IN':
131.89 MHz (PASS at 25.00 MHz)
Info: Max frequency for clock              '$glbnet$hdmi_pll_CLKOP':
379.08 MHz (PASS at 250.00 MHz)
Info: Max frequency for clock  'io_hdmi_tmds_clk_p$TRELLIS_IO_OUT':
42.65 MHz (PASS at 25.00 MHz)
Info: Max frequency for clock '$glbnet$io_lan_txclk$TRELLIS_IO_IN':
104.96 MHz (PASS at 25.00 MHz)
Info: Max frequency for clock           'io_clkusb$TRELLIS_IO_OUT':
116.02 MHz (PASS at 12.00 MHz)
Info: Max frequency for clock  '$glbnet$io_clkcore$TRELLIS_IO_OUT':
65.55 MHz (PASS at 60.00 MHz)
...

Info: Program finished normally.

ecpbram -v -i bin/KarnixSOCTopLevel_random_25F.config -o
bin/KarnixSOCTopLevel_25F.config -f
../../../KarnixSOCTopLevel_random.hexx -t
../../../src/main/c/karnix/karnix_bootloader/build/karnix_bootloader
.hexx
Padding to_hexfile from 2769 words to 18432
Loaded pattern for 32 bits wide and 18432 words deep memory.
Extracted 1152 bit slices from from/to hexfile data.
ecppack --svf bin/KarnixSOCTopLevel_25F.svf
bin/KarnixSOCTopLevel_25F.config bin/KarnixSOCTopLevel_25F-
karnix_bootloader.bit

rz@devbox:~/RISCV/KarnixSOC/scripts/KarnixSOC/ECP5-25F_karnix_board$ ls -l
bin/*.bit
-rw-rw-r-- 1 rz rz 709859 Oct 16 21:22 bin/KarnixSOCTopLevel_25F-
karnix_bootloader.bit
```

// Информация о проекте СнК «Карно» (KarnixSoC)

FPGA SYSTEMS 2025

1. Проект платы «Карно» выполненный в САПР KiCAD (в т.ч. герберы, схемы в PDF, BoM):
https://github.com/Fabmicro-LLC/Karnix_ASB-254

2. Ситезируемая СнК «Карно» на SpinalHDL:
https://github.com/pointcheck/KarnixSOC

3. Статья на Хабре «Разработка цифровой аппаратуры нетрадиционным методом: Контроллер USB 1.0 на SpinalHDL».
https://habr.com/ru/articles/908828/

Ищу единомышленников и энтузиастов для развития проекта и портирования на другие ПЛИС!