RFID 原理与应用实验

文中所有的引用参考，均指：“OURS-RIFD-RP 实验指导书”

第三章 RFID 原理机实验，其中 3.4 ”ISO15693 协议标签操作”，使用 HFPRIN.exe 程序执行每个命令，不涉及编码、调制，也不涉及防碰撞算法；可以认为是在展示采用 FPGA 实现的 RFID Transceiver 能力：实现了编码、调制等能力！从实验角度看，和 4.2 相同，不做。

3.1 在介绍原理图以及通信协议；

3.3 是编码和调制；开设编码和调制实验；

1 编码与调制

1.1 实验目的、手段

目的：掌握编码与调制知识，加深对课堂理论学习的理解；

手段：使用示波器，观察解调信号波形；

备注：只能观察 ISO15693 协议的信号；

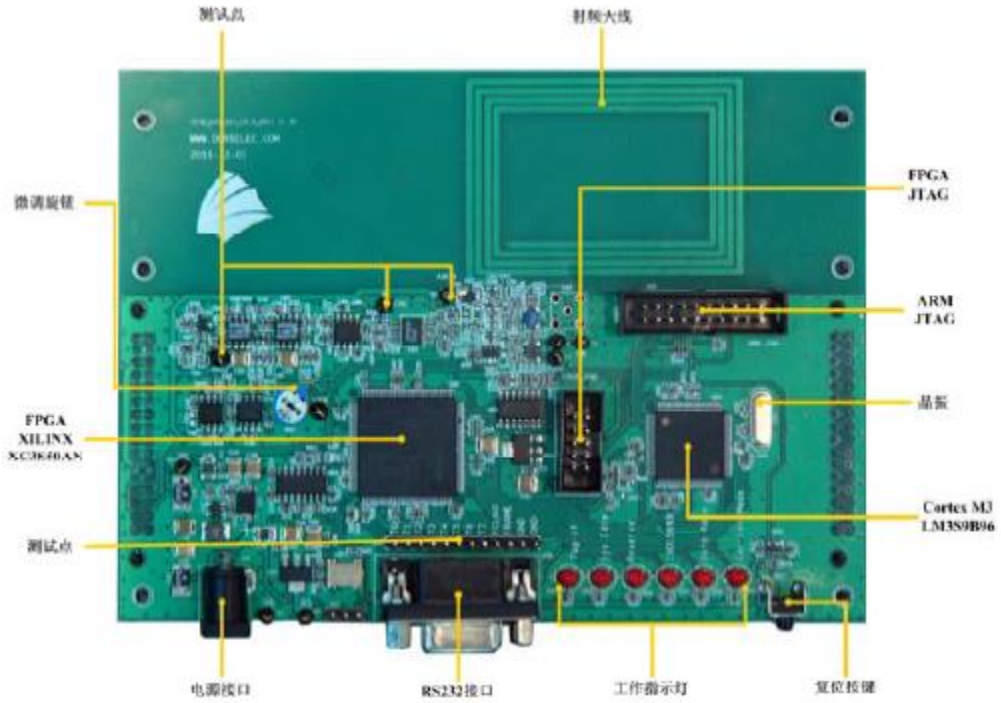
1.2 实验环境

1.2.1 硬件环境

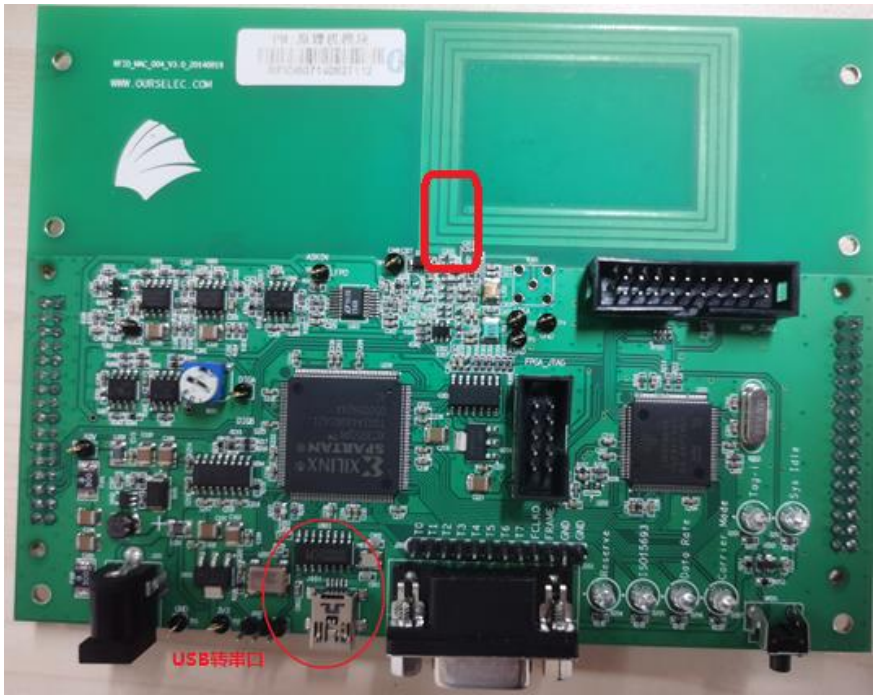
Page 41

CHAR 2.1.3，part 4.

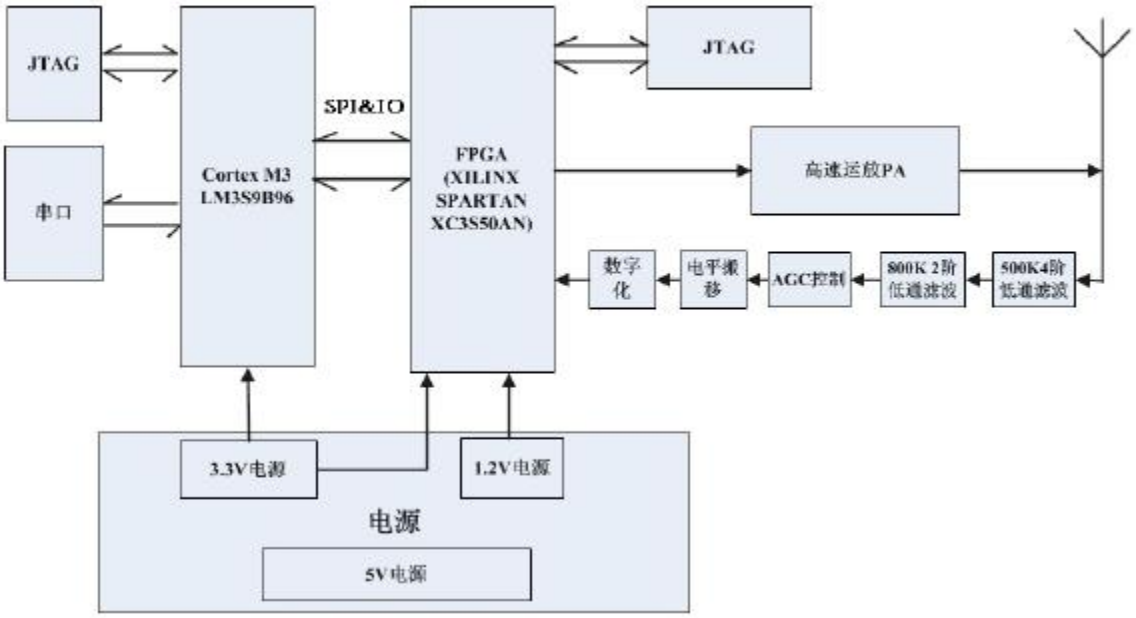
板卡（需要核实，看 RS232 口的 DB9 是否已经换掉了）：有 USB 转串口.



下图是实际使用电路板，下端增加了 USB 口，其中后文提到的电容 C315 的位置为上端红线圈内。



HF 原理机系统框图



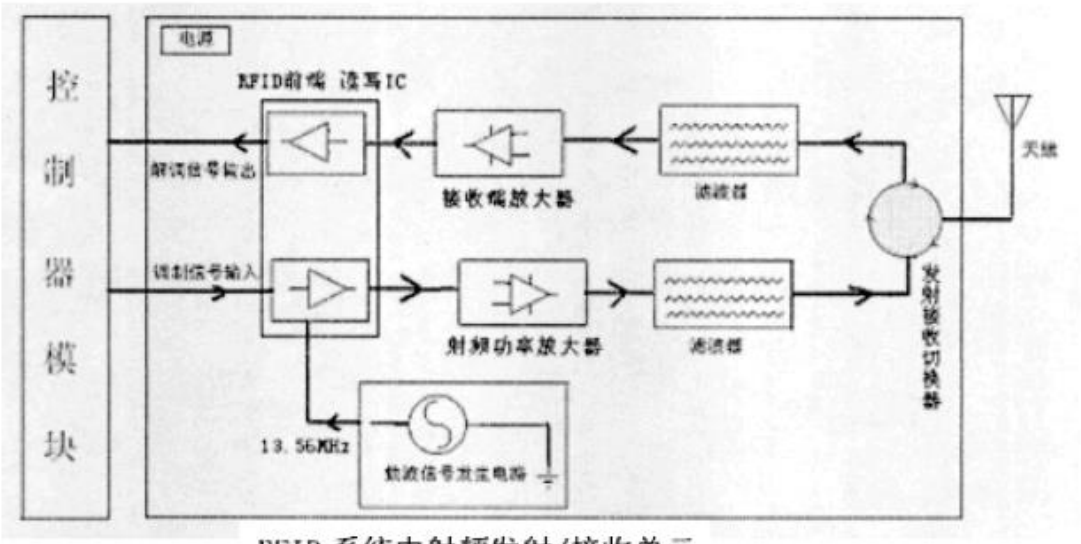
硬件环境主要芯片两个：

MPU：LM3S9B96，主要作用：1。通过 RS232 口，和操作显示终端（比如 PC 机）通信，2，从串口接收控制端发来的命令，发送到 FPGA 中；其中一些命令控制 FPGA 运行，比如：运行哪种协议；另一些命令由 FPAG 实现组帧、编码、调制后，发送给 VICC，并接收 VICC 的结果，比如 15693 命令；

RFID 协议的编码、调制等由 FPGA 实现；MPU 和 FPGA 之间通过 SPI 总线连接。MPU 发送命令，FPGA 返回命令执行的结果.

Page 65, 3.1.1 中给的框图：

以分立部件的形式展现 RFID 的读写原理，包括发射信号调制、解调等，可以通过示波器观察测试点波形，加深对原理的理解。



RFID 系统中射频发射/接收单元

测试点：

本振、发送载波、调制载波、射频功率放大信号、接收射频信号、接收调制信号，接收解调信号。原理机的控制模块采用另外的单片机 Cortex M3 实现，通过串口与网关或 PC 机通信。

但指导书缺少在电路板上标注测试位置的位置。

Part 3.1.2, Page66 中，如下描述，但实际只支持 ISO/IEC 15693 协议。另外：明确：成帧、编码、调制在 FPGA 实现。

根据上图，在设计中拟实现了如下硬件性能：

由阅读器到应答器通路：

- 实现了修正密勒码、NRZ 码以及 PPM 码（2）的编码（FPGA 逻辑实现）。
- 实现了 0%、10%、20%、100%等 4 档 ASK 调制（FPGA 逻辑实现）。
- 实现了 PPM 的成帧控制（FPGA 逻辑实现）。
- 实现了可控的阅读器输出功率选择（FPGA 逻辑实现）。

由应答器到阅读器通路：

- 实现基本的前置放大、滤波电路。
- 实现非相干的解调控制。
- 实现与副载波相关的曼彻斯特码、NRZ 码的解码（FPGA 实现）。

与数据控制有关的电路采用 Cortex M3 内核的 CPU+FPGA 联合处理的方法实现，CPU 实现对于 RFID 的协议控制及与上位机的接口，FPGA 完成实际的数据输入输出（包括编解码）及硬件电路的控制。

1.2.1.1 LM3S9B96

CPU: LM3S9B96(ARM Cortex-M3).

LM3S9B96 属于 TI Stellaris 系列(Stellaris® LM3S9B96 Microcontroller)MPU，datasheet 长达 1281 页，是 TI 公司基于 ARM Cortex-M3 的 32 位 MCU，CPU 工作频率 80MH，哈佛结构,片内集成了 256KB FLASH 和 96KB SRAM,在 TI 官网芯片的分类中,controllers 包括 ARM-based microcontrollers 和 ARM-based processors 两类,前者分类下列有一些芯片,比如:MSP432E401Y，CPU 为 Arm Cortex-M4F，TI 把 Cortex-Mx 系列定位为单片机（不带 OS），ARM-based processors 下的 ARM 使用的是 A 系列，比如 Arm Cortex A9/8/15/53 等。

可惜在 TI 的官网上找不到 LM3S9B96 芯片的 datasheet 了(已经不生产了)。

在 Arm 系列芯片中，“Cortex-A 属于高性能处理器，定位于物联网，路由器设备等，**Cortex-M 系列主打低功耗低成本和较强性能设备**，Cortex-R 系列主打低时延实时应用，适用于无线基站，手机基带等。

Cortex-M 系列中又分为 Cortex-M0 ， Cortex-M3 ， Cortex-M4 ， Cortex-M7 等架构，常见的 STM32F103 单片机就是意法半导体 (ST) 公司基于 Cortex-M3 架构生产的处理器，这款处理器成本低廉而性能强大，最重要的是它在全世界范围有着非常广泛的应用，教学资料随处可见，非常适合用于嵌入式入门，而且 Cortex-M3 的架构设计非常经典，只要掌握了它，要转到其他嵌入式平台也是非常简单的。

LM3S9B96 datasheet overview 部分，来自于网络（截图）。

Architectural Overview

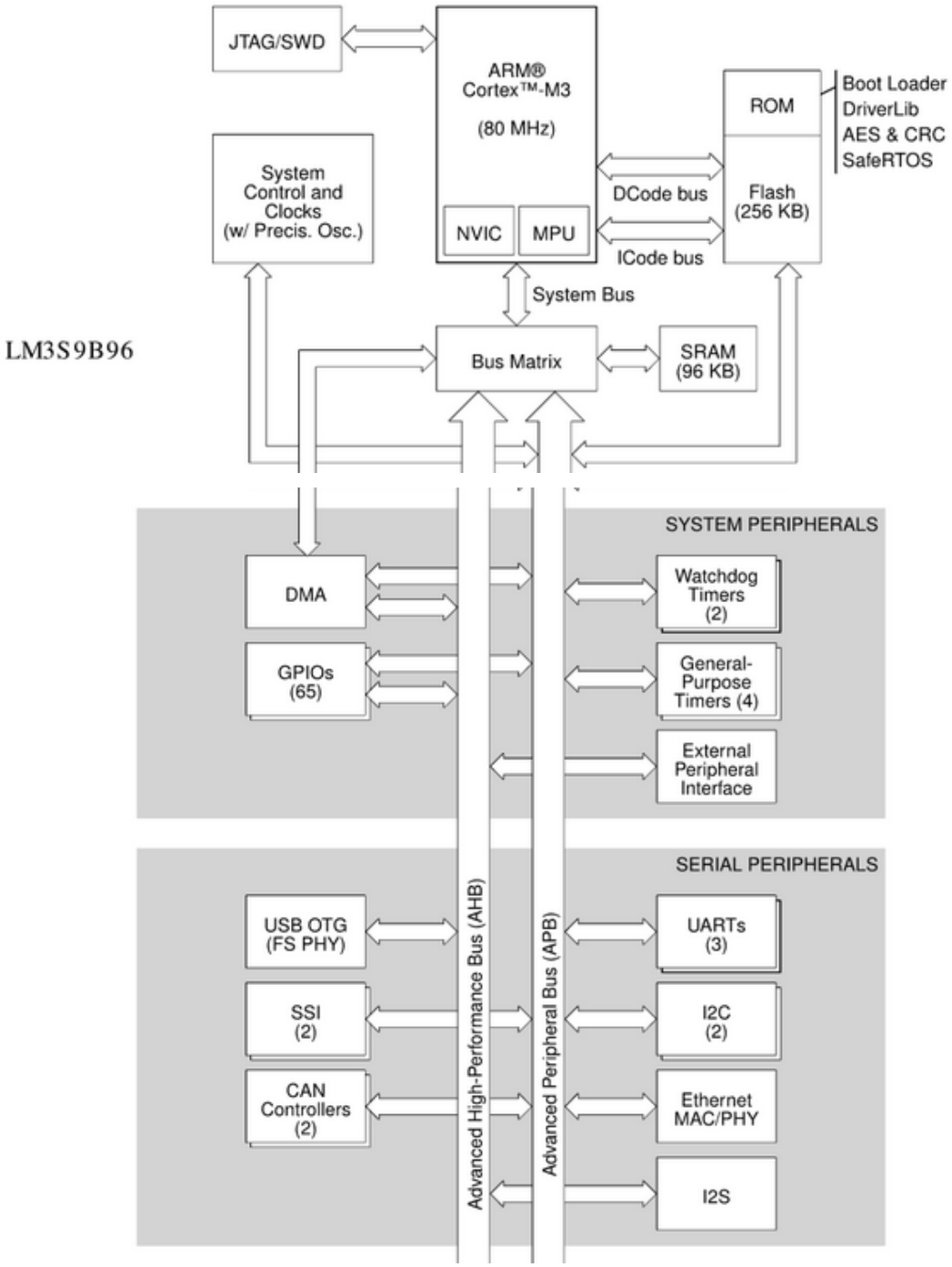
Texas Instruments is the industry leader in bringing 32-bit capabilities and the full benefits of ARM® Cortex™-M3-based microcontrollers to the broadest reach of the microcontroller market. For current users of 8- and 16-bit MCUs, Stellaris® with Cortex-M3 offers a direct path to the strongest ecosystem of development tools, software and knowledge in the industry. Designers who migrate to Stellaris benefit from great tools, small code footprint and outstanding performance. Even more important, designers can enter the ARM ecosystem with full confidence in a compatible roadmap from $1 to 1 GHz. For users of current 32-bit MCUs, the Stellaris family offers the industry's first implementation of Cortex-M3 and the Thumb-2 instruction set. With blazingly-fast responsiveness, Thumb-2 technology combines both 16-bit and 32-bit instructions to deliver the best balance of code density and performance. Thumb-2 uses 26 percent less memory than pure 32-bit code to reduce system cost while delivering 25 percent better performance. The Texas Instruments Stellaris family of microcontrollers—the first ARM Cortex-M3 based controllers—brings high-performance 32-bit computing to cost-sensitive embedded microcontroller applications. These pioneering parts deliver customers 32-bit performance at a cost equivalent to legacy 8- and 16-bit devices, all in a package with a small footprint.

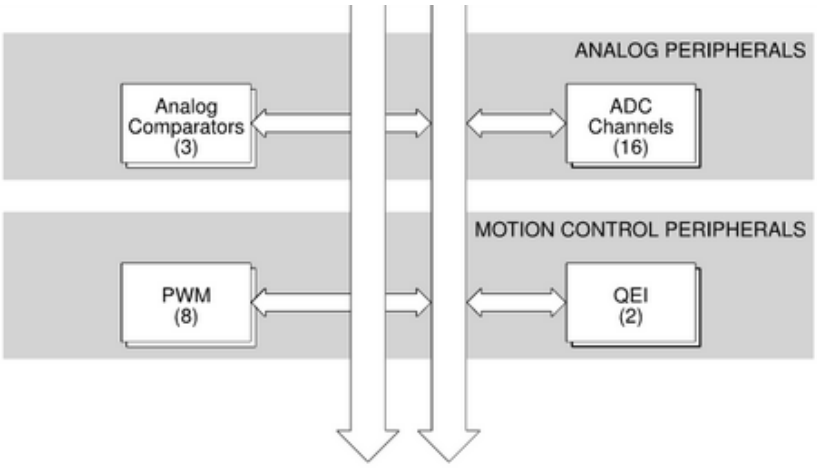
The LM3S9B96 microcontroller has the following features:

- ARM Cortex-M3 Processor Core
  - 80-MHz operation; 100 DMIPS performance
  - ARM Cortex SysTick Timer
  - Nested Vectored Interrupt Controller (NVIC)
- On-Chip Memory
  - 256 KB single-cycle Flash memory up to 50 MHz; a prefetch buffer improves performance above 50 MHz
  - 96 KB single-cycle SRAM
  - Internal ROM loaded with StellarisWare® software:
    - Stellaris Peripheral Driver Library
    - Stellaris Boot Loader
    - SafeRTOS™ kernel
    - Advanced Encryption Standard (AES) cryptography tables
    - Cyclic Redundancy Check (CRC) error detection functionality
- External Peripheral Interface (EPI)
  - 8/16/32-bit dedicated parallel bus for external peripherals
  - Supports SDRAM, SRAM/Flash memory, FPGAs, CPLDs
- Advanced Serial Integration
  - 10/100 Ethernet MAC and PHY with IEEE 1588 PTP hardware support
  - Two CAN 2.0 A/B controllers
  - USB 2.0 OTG/Host/Device
  - Three UARTs with IrDA and ISO 7816 support (one UART with full modem controls)
  - Two I^2C modules
  - Two Synchronous Serial Interface modules (SSI)
  - Integrated Interchip Sound (I^2S) module

- System Integration
  - Direct Memory Access Controller (DMA)
  - System control and clocks including on-chip precision 16-MHz oscillator
  - Four 32-bit timers (up to eight 16-bit), with real-time clock capability
  - Eight Capture Compare PWM pins (CCP)
  - Two Watchdog Timers
    - One timer runs off the main oscillator
    - One timer runs off the precision internal oscillator
  - Up to 65 GPIOs, depending on configuration
    - Highly flexible pin muxing allows use as GPIO or one of several peripheral functions
    - Independently configurable to 2, 4 or 8 mA drive capability
    - Up to 4 GPIOs can have 18 mA drive capability
- Advanced Motion Control
  - Eight advanced PWM outputs for motion and energy applications
  - Four fault inputs to promote low-latency shutdown
  - Two Quadrature Encoder Inputs (QEI)
- Analog
  - Two 10-bit Analog-to-Digital Converters (ADC) with 16 analog input channels and a sample rate of one million samples/second
  - Three analog comparators
  - 16 digital comparators
  - On-chip voltage regulator
- JTAG and ARM Serial Wire Debug (SWD)
- 100-pin LQFP and 108-ball BGA package
- Industrial (-40°C to 85°C) Temperature Range

Figure 1-1. Stellaris LM3S9B96 Microcontroller High-Level Block Diagram





1.2.1.2 FPGA 内部设计

Part 3.1.7, page 71 部分，说明 FPGA 实现的功能：

VCD→VICC：

SOF 设计；

EOF 设计；

256 取 1、4 取 1 逻辑设计；

VICC→VCD：

单副载波的 SOF 检测；

双副载波的 SOF 检测；

单副载波的 EOF 检测；

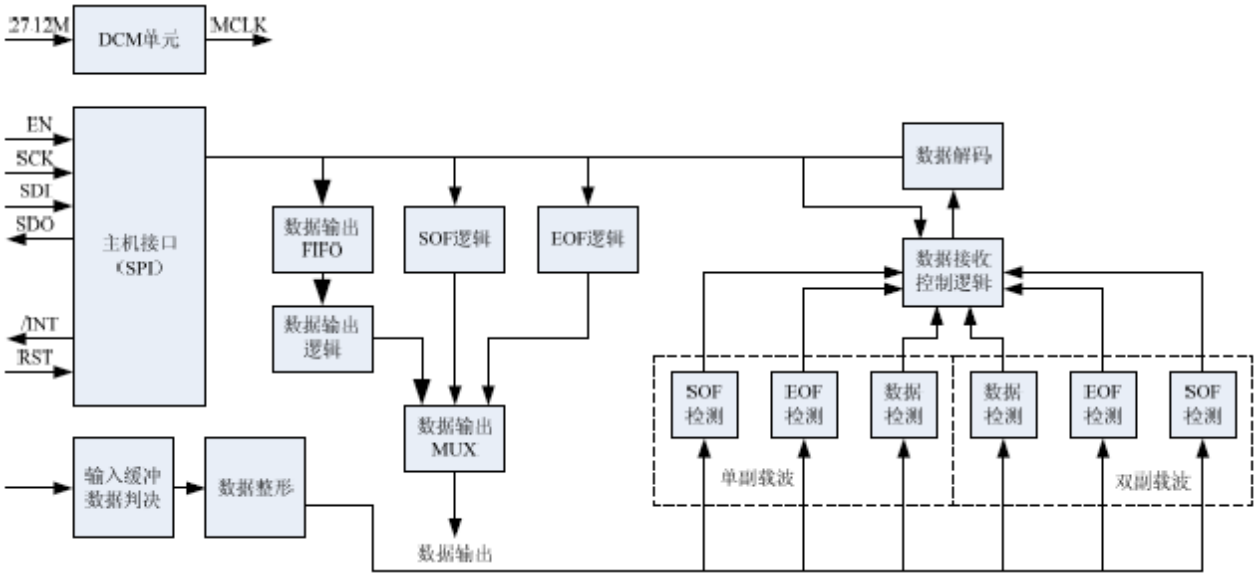
双副载波的 EOF 检测；

单副载波的数据检测；

双副载波的数据检测；

可见：成帧，发送方向（脉冲位置编码），接收方向解码（曼彻斯特码）、解帧（副载波解调）

FPGA 内部框图：



但：上图在数据解码后，应该到接收 FIFO 部分

Part 3.1.7，page 73.

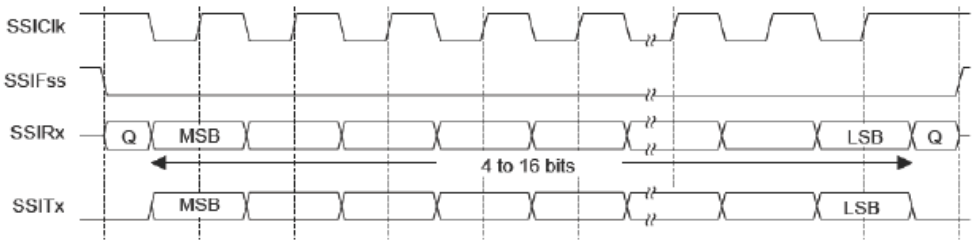
MPU 通过 SPI 总线，向 FPGA 内部的寄存器写命令。FPGA 在收到命令后，执行，并返回结果。

SPI 接口定义如下：

FPGA 主机接口协议（SPI 接口）

FPGA 与主机采用标准 SPI 接口方式对接，见下图：

Figure 14-9. Freescale SPI Frame Format with SPO=1 and SPH=1



Note: Q is undefined.

SPI 接口的规定如下：

★使用 8 位数据（字节）方式。
★SPI 接口的时钟频率不得大于 2M（建议 2M）。
★可一次连续传输多个字节（>=1）。
★首字节为系统行为控制字节（详见下表）。
★连续传输数据按地址自增方式进行控制。

其中：注意首字节为控制字节，就是 1.2.3 中提到的“CMD”.

首字节定义如下：

BIT.7	BIT.[6:4]	BIT.3	BIT[2:0]
EIF	REQ_INX[2:0]	RST	ADDR[2:0]
0	0	0	0

“EIF”：内部、外部操作标志，0 = FPGA 内部操作； 1 = 对 RFID 接口操作。

“REQ_INX[2:0]”：内部、外部操作命令代码，

针对于 EIF = 0。

000：内部寄存器写；

001：内部寄存器读；

010：内部 FIFO 写（最大 64 字节）；

011：内部 FIFO 读（最大 64 字节）；

1xx：针对 CMD 操作（无后续数据）

此处CMD是啥？

针对于 EIF = 1。 此处应该是向RFID Card 发送命令

000：从 RFID 发送一条指令（发送完毕后自动切换到接收）；

001：从 RFID 发送“EOF”；

010：启动一个延迟操作；

其他：保留；

“RST”：模块软复位控制，高有效。直接连接到模块复位管脚，正常使用时需置为无效状态。

“ADDR[2:0]”：（仅在 EIF = 0 时，且对内部寄存器操作时有效）地址位，使用 3bit 的字节地址，制定要操作的寄存器的首地址（字节地址）。当操作超过一个字节时，按地址自增方式处理（地址有效范围为 0~7，超出范围采用回绕方式处理，操作者需要自行处理越界问题）。

注意： 内部

ADDR[2:0]目前为保留应用，也就是说针对~~北部~~ FIFO 或寄存器进行读写时，始终是从地址 0 开始的。并且针对内部寄存器写操作，一次必须写固定 64bit（8 字节）。

当只想输出 SPI 的 CMD 字节数据时（如生成内部“RESET”信号），需要将“EIF=0”；“REQ_INX[2:0]=1xx”。

1．ADDR[2:0]没用，只用于访问内部寄存器，且每次一定读/写 8 个寄存器。

2．特别：

a） Cmd=0x00，写内部寄存器；

b） Cmd=0x10，读内部寄存器；

c） Cmd=0x20，写内部 FIFO；

d） Cmd=0x30，读内部 FIFO；

2．上面特殊的，似乎只有 RFID 模式？

EIF = 1 时，是用于 0x20/21 类指令？如果是用于 0x20,0x21 类指令，那么 Tx FIFO 又有啥用？

寄存器定义如下：

已经定义的命令地址如下：

序号	名称	地址	定义	定义
1	SYS_CTRL	0x0	W	系统进程控制。
2	SYS_STUS	0x0	R	系统状态。
3	DATA_LEN	0x1	W	本次 RFID 数据（FIFO）操作的数据长度。操作的首地址默认为 0。有效值为 1~64（代表 1~64 个字节操作）
4	REV_DNUM	0x1	R	RFID 已经接收的数据长度。默认=0，RFID 数据接收关联 FIFO 的加操作；SPI 数据读操作关联 FIFO 的减操作。
5	15693_CTL	0x2	R/W	15693 协议控制。
6	14443A_CTL	0x3	R/W	14443A 协议控制（保留）。
7	14443B_CTL	0x4	R/W	14443B 协议控制（保留）。
8	DLY_NUML	0x5	R/W	系统延时控制字节，t=36.873nS。
9	DLY_NUMH	0x6	R/W	
10	RSV_bits	0x7	R/W	

“SYS_CTRL”寄存器定义：

BIT.7	BIT.6	BIT[5:4]	BIT.3	BIT.2	BIT.1	BIT.0
RSV	RSV	PINX[1:0]	TXEN	TXEI	RXEN	RXEI
0	0	0	0	0	0	0

“PINX[1:0]”：当前使用协议，00=ISO15693（默认）；01=ISO14443A；10=ISO14443B;
11=保留。

“TXEN”：TX 工作方式控制，高有效。
“TXEI”：TX 工作方式中断输出允许控制，高有效。
“RXEN”：RX 工作方式控制，高有效。
“RXEI”：RX 工作方式中断输出允许控制，高有效。

寄存器定义了 TXEI 和 RXEI，实际应用需要软件配合。

“SYS_STUS”寄存器定义：

BIT.7	BIT.6	BIT[5]	BIT[4]	BIT.3	BIT.2	BIT.1	BIT.0
RSV	RSV	RSV	TBUSY	RBUSY	DBUSY	FWERR	FRERR
0	0	0		1	1	0	0

“TBUSY”：RFID 输出运行忙标志，高有效。
“RBUSY”：RFID 读运行忙标志，高有效。
“DBUSY”：RFID 延时运行忙标志，高有效。
“FWERR”：数据 FIFO 写操作错误，高有效。
“FRERR”：数据 FIFO 读操作错误，高有效。

“15693_CTL”寄存器定义：

BIT.7	BIT.6	BIT[5:4]	BIT.3	BIT.2	BIT.1	BIT.0
RSV	RSV	RSV	RSV	DRAT	SUBW	DCOD
0	0	0	1	0	0	0

“DCOD”：发送数据编码控制，0 = 256 取 1；1 = 4 取 1。
“SUBW”：副载波控制，0 = 单副载波；1 = 双副载波。
“DRAT”：VICC 数据率，0 = 低数据率； 1 = 高数据率。

这些定义了 15693 协议中，VCD 访问 VICC 时，使用的编码和调制方式。

从上面看：如果向 VICC 发送数据，需要：

1．写 15693_CTL 寄存器，确定编码，调制等方式；

2．写 DTAT_LEN 寄存器，说明本次向 FIFO 写数据的长度；

3．写 FIFO，数据（不包括 SOF 和 EOF）

4．写 SYS_CTL 寄存器，TXEN & RXEN；

5．读 REV_DNUM 寄存器，等待数据到达；

6．读 FIFO；

其中：读写寄存器每次必须是 8 个。

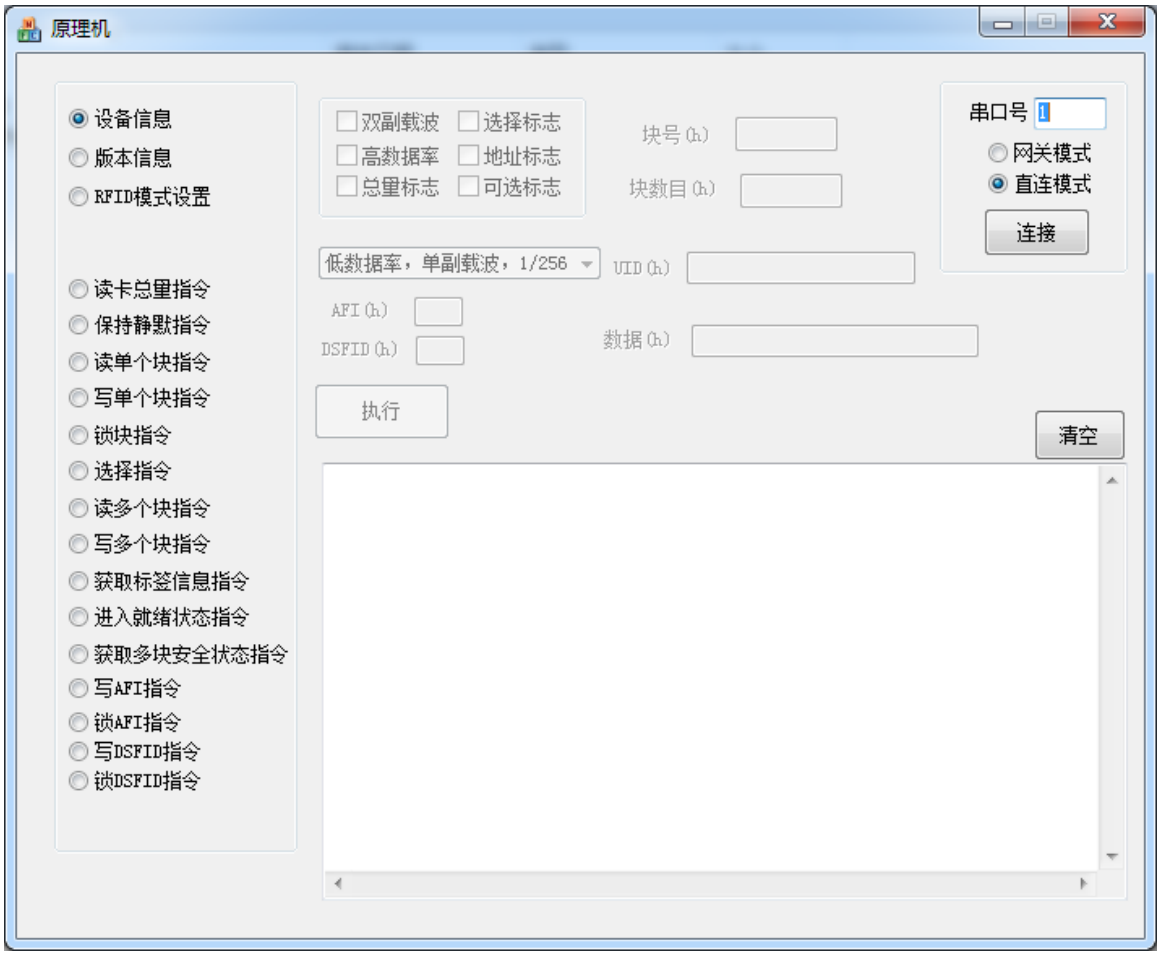
这样看，MPU 在 SPI 之上，实现了一些协议操作。

1.2.2 软件环境

PC 端，windows 下，运行程序：

1．串口调试助手，展示基础能力；

2．展示协议部分：HFPRIN.exe，不需要额外的 DLL。界面如下：



1.2.3 PC 和硬件通信协议

Part 3.1.8, page 76.

PC 端和硬件采用串口连接，通信协议如下：

2. 原理机通讯协议

原理计使用串口与上位机进行通讯，在通讯过程中，原理机始终作为从机，并严格执行问答式通讯，关于通讯协议一般要求如下：

- ★ 使用串口进行通讯，串口配置为（115200bps，8 数据位，1 起始/停止位，无校验）。
- ★ 使用应答式通讯，通讯始终由上位机发起，从机应答。
- ★ 通讯超时错定义为 10mS，主机下传数据字节间隔不能大于 10mS。从机上传数据数据间隔同样不大于 10mS。
- ★ 从机相应主机请求时间一般小于 100mS（特殊指令除外，详见下文）。
- ★ 从机在未处理完上一条指令前不接收新的指令。

主机下传指令一般格式如下：

字节 1	字节 2	字节 3	字节 n	字节 n+3
HEAD	LEN	CODE	DATA	CRC
0x55				

从说明看，LEN 和 CODE 一定是存在的。CRC 是对 DATA 的累加和． LEN 不包括 CODE 和 CRC．

“LEN”可以为 0。当“LEN > 0”时，表示后跟“LEN”个数据及一个 CRC 字节。当“LEN=0”时，DATA 以及 CRC 不存在。

“CRC”实际上用累加和代替。 不包括code/cmd

从机上传数据（应答）一般格式如下：

字节 1	字节 2	字节 3	字节 n	字节 n+3
HEAD	LEN	CODE	DATA	CRC
0xAA				

通讯中用到的指令如下：

序号	命令	数据长度		指令解释
		下行	上行	
1	0x00	0	16	设备信息：固定返回“www.ourselec.com”字符串
2	0x01	0	8	版本信息：返回“Ver:x.xx”格式字符串。
3	0x02	1	0	RFID 模式设置字节（见下例）。
4	0x10	9	0	SPI 接口的写（透明写），数据定义详见“5.2.3”，**注意：针对 FPGA 内部寄存器写操作要求一次操作固定写 8 字节数据。** 针对 FPGA 内部 FIFO 写操作要求预先使用寄存器写设置 FIFO 写操作的数据个数。
5	0x11	2	n	SPI 接口的读（透明读），数据定义详见“5.2.3”，**注意：针对 FPGA 内部寄存器读操作要求一次操作固定写 8 字节数据。** 针对 FPGA 内部 FIFO 读操作要求预先使用寄存器写设置 FIFO 读操作的数据个数。
6	0x20	n	x	读卡总量指令。控制 FPGA 发起一次 VCD→VICC 的请求。
7	0x21	n	x	除读卡总量外其他指令，控制 FPGA 发起一次 VCD→VICC 请求，并返回数据。

所谓透明，是MPU不解释具体含义，把收到的内容直接传输给FPGA，提供了PC端对FPGA操作能力。具体数据内容组织由计算机端的软件负责

此处命令，就是 CODE。

其中：0x10，0x11，所谓透明写/读，是 MPU 不解释其内容，把这些数据直接通过 SPI 总线发送给 FPGA，由 FPGA 处理。其中包括了 SPI 总线的第一个字节 CMD.

相对应的，0x20,0x21，是串口数据发送给 MPU，由 MPU 控制 FPGA（当然，也是通过 SPI 总线），执行 RFID 操作，其要完整实现一个 15693 协议发送（request）和接收(response)操作过程，并把接收到的数据返回到 PC；其中 0x20，即 inventory 命令，由 FPGA（大概率是 MPU）一次执行结束。

RFID 模式设置字节定义见““**15693_CTL**”寄存器定义” 编码与调制控制。

SPI 写操作的数据定义：

“CMD 字节”,“数据（n 个字节，针对寄存器写，n=8；针对 FIFO 写，个数为预先设定的数据个数）” 0x0x00

需要注意的是，执行 FPGA 的 FIFO 写操作操作，FPGA 不会检查写数据的个数。

指令应答为“0xaa”、“0x00”、“0x10”。

SPI 读操作的数据定义： 0x10

串口下行数据为 2 字节，依次为“CMD 字节”,“数据个数字节”。

串行数据上行数据个数为下行时数据个数字节指定的数目（针对 FPGA 的寄存器读，数据个数固定为 8）。 “CMD字节”,见后面的例子

后面的例子，可以说明；

1.设置 RFID 基本工作模式

在进行 RFID 操作前，需要使用本指令设置 RFID 工作模式。后续的控制数据必须与设置的模式相一致。

寄存器地址从 0~7 数据依次为 0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88

串口输出数据：（PC->原理机）

0x55,0x01,0x02, mode, mode

“mode”字节要连续发两遍，第 2 遍的含义是累加和校验。

从机返回：（原理机->PC）

0xaa,0x00,0x02

MODE 定义：

0x00 ： 代表低数据率，单副载波，输出数据位“256 取 1 方式”。
0x01 ： 代表低数据率，单副载波，输出数据位“4 取 1 方式”。
0x02 ： 代表低数据率，双副载波，输出数据位“256 取 1 方式”。
0x03 ： 代表低数据率，双副载波，输出数据位“4 取 1 方式”。
0x04 ： 代表高数据率，单副载波，输出数据位“256 取 1 方式”。
0x05 ： 代表高数据率，单副载波，输出数据位“4 取 1 方式”。
0x06 ： 代表高数据率，双副载波，输出数据位“256 取 1 方式”。
0x07 ： 代表高数据率，双副载波，输出数据位“4 取 1 方式”。

例如： 55 01 02 05 05

将 RFID 模式设置为高数据率，单副载波，输出数据位“4 取 1 方式”。

55 01 02 07 07

将 RFID 模式设置为高数据率，双副载波，输出数据位“4 取 1 方式”。

此处 mode 定义显然和 FPGA 内部寄存器定义的模式不同，显然是 MPU 解释的模式。MPU 将把该模式翻译

成 FPGA 的模式(MPU 通过 SPI，写 15693_CTL 寄存器，SYS_CTRL 寄存器)。

当然，MPU 也要遵守 SPI 协议，比如为了修改 15693_CTL 寄存器，需要做“读/修改/写”操作，读全部 8 个寄存器，修改第 5 个寄存器，再把这 8 个数据全部回写到 FPGA 中。

2.SPI 寄存器写

寄存器地址从 0~7 数据依次为 0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88

串口输出数据： 这个0x00是啥?

0x55,0x09,0x10,0x00, 0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88, 0x64

从机返回：

0xaa,0x00,0x10 返回的是命令字?

要写 FPGA 寄存器，则需要 CODE 0x10，寄存器必须 8 个一起操作，再加命令字 CMD，所以长度为 9.

此处 0x00 表示写寄存器。

Bit7 = 0，表示操作 FPGA 内部；

Bit 6-4, 000：内部寄存器写；

下面是 SPI 第一个字节(CMD)定义：

BIT.7	BIT.[6:4]	BIT.3	BIT[2:0]
EIF	REQ_INX[2:0]	RST	ADDR[2:0]
0	0	0	0

3.SPI 寄存器读

读寄存器地址从 0~7 的 8 个数（固定读）

串口输出数据：

0x55,0x02,0x11,0x10, 0x08,0x18 （固定读 8 个字节）

从机返回：

0xaa,0x08,0x11,0xN0, 0Xn1, 0xN2, 0xN3, 0xN4, 0xN5, 0xN6, 0xN7, 0xCRC

（0xN0~0xN7 为寄存器地址从 0~7 的数据，最后是累加校验和）

CODE = 0x11：从 SPI 接口读；

0x10：0,001,0,000； 001：读 FPGA 内部寄存器；

0x08:长度必须为 8 字节，但也需要给出长度。

4.FPGA 的 FIFO 输出（写，假定一次写 8 个字节）

串口输出数据（首先使用寄存器写输出 FIFO 操作数据个数）：
0x55,0x09,0x10,0x00,0xXX, 0x08, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX, 0xCRC
从机返回：
0xaa,0x00,0x10

串口输出数据（依次输出 D0~D7 数据）：
0x55,0x09,0x10,0x20,0xD0, 0xD1, 0xD2, 0xD3, 0xD4, 0xD5, 0xD6, 0xD7, 0xCRC
从机返回：
0xaa,0x00,0x10

分两次，

先用 0x00，写 0x08 到地址 1（DATA_LEN），表示写 FIFO 的长度（如果要这样写，还需要先执行读操作。）；

然后用 0x20，表示写 FIFO，长度在第一次操作中已经说明了。

0，010，0，000：010 表示写内部 FIFO

FPGA 内部 FIFO 最大 64 字节。

此处标题：FIFO 输出，含义是 FIFO 中的数据会自动输出，不需要那个 TXEN 吗？

5.FPGA 的 FIFO 输入（读，假定一次读 8 个字节）

串口输出数据（首先使用寄存器写输出 FIFO 操作数据个数）：
0x55,0x09,0x10,0x00,0xXX, 0x08, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX, 0xXX, 0xCRC
从机返回：　　应该是错的！
0xaa,0x00,0x10

串口输出数据（读 FIFO 数据）：
0x55,0x02,0x11,0x30, 0x08,0x38　（读 8 个字节）
从机返回：
0xaa,0x08,0x11,0xN0, 0xN1, 0xN2, 0xN3, 0xN4, 0xN5, 0xN6, 0xN7, 0xCRC

从 FPGA 内部读，分两次：

1．设置长度；

2．开始读：

0x30：0，011，0，000；011：读 FIFO；

读的长度，在 0x10 中，写寄存器中已经定义了。

6.FPGA 发起 RFID 请求

串口输出数据（需要发出的数据将使用 FIFO 写指令预先写好）：
0x55,0xXX,0x20,
0xA,0xB, 0xC, 0xD0, ~0xDx, 0xE , 0xCRC（红色为发到 RFID 的数据，依次为标志、命令、MASK 长度、MASK 值（0~8 字节）、CRC16（两字节））

从机返回：
0xaa,0x00,0x20

6：CODE = 0x20，读卡总量命令，这个是 ISO15693 协议的 inventory 命令。
红色部分是 inventory 命令
6 只是一个格式， 7 才是具体例子。

不清楚，执行这个命令后，MPU 是否会实现点名操作：即多次使用 inventory + EOF，把一个或者多个 VICC 识别出来。

7.控制 FPGA 发起 RFID 卡的总量获取请求

串口输出数据（需要发出的数据将使用 FIFO 写指令预先写好）：
0x55, 0x05, 0x20, 0x06, 0x01, 0x00, 0xcd, 0x09, 0xdd
“0x20”为控制命令字节（本文中定义）
“0x06”、“0x01”、“0x00”为 RFID 协议中的控制，代表使用单副载波，高数据率。
“0xcd”、“0x09”为 RFID 协议中的 CRC16 的值，需要预先算好。
“0xdd”为本控制的累加校验和（0x06, 0x01, 0x00, 0xcd, 0x09 等 5 个数相加）

从机返回（两种可能）：
1. 0xaa,0x00,0x20（未找到卡）
2. AA 0C 20 00 EE 51 D1 78 14 00 00 07 E0 1E 6E 0F
   其中的 “0x0c”为本指令上传数据个数（00 EE 51 D1 78 14 00 00 07 E0 1E 6E）。
   上传数据的第 3~10 字节为 RFID 卡的 ID 号（51 D1 78 14 00 00 07 E0，低字节在前）
   “0x1e”、“0x6e”为上传的 CRC16 数值，低字节在前，
   最后一字节为本指令添加的累加和校验。

数据：0x20, 0x06, 0x01, 0x00, 0xcd, 0x09
0x06：表示：0000,0110;
Bit1 = 0, 单副载波;
Bit2=1, 高速率;
Bit3 = 1, 表示 inventory 命令;
Bit 4 = 0, 无扩展;
Bit 5 =0, 没有 AFI;
Bit 6 = 0, 16 slots;

Bit 7 = 0,

Bit 8 RFU;

0x01: 15693 命令：inventory.

0x00: MASK 长度.

0xCD 0x09 CRC16

所以，这是执行 inventory 命令；

返回：长度 0x0C 对的。

返回：0x00, 0xEE 没有给出说明：

0x00：就是 VICC->VCD 的响应标志?：0x00;

0xEE 又是啥呢？

实际 inventory，需要多次完成。。。那么，可以用此实现防碰撞算法吗？来的及吗？

就是用于实现放碰撞算法的。

Part 3.4.5, Page 125 的例子，也没有说明：前面 2 个字节是啥？

但：这两个例子都说明了：MPU 执行了完整的防碰撞过程：因为返回了 VICC ID.

8.控制 FPGA 发起 RFID 卡的块读取请求

读第 1 个块，返回 8 字节数据（使用前条指令定义好的读写设置，如数据率、单/副载波等）。

串口输出数据（需要发出的数据将使用 FIFO 写指令预先写好）：

0x55, 0x05, 0x21, 0x03, 0x20, 0x01, 0x12, 0x1b, 0x51

“0x21”为控制命令字节（本文中定义）

“0x03”、“0x20”、“0x01”为 RFID 协议中的控制，代表使用双副载波，高数据率。

从机返回（两种可能）：

3. 0xaa,0x00,0x21（未找到卡）
4. 找到卡返回 AA 07 21 00 77 77 77 77 3A 74 8A
   其中的 “0x07”为本指令上传数据个数（00 77 77 77 77 3A 74）。
   上传数据的第 1~4 字节为 RFID 卡的块 1 的 4 字节数据（77 77 77 77，低字节在前）
   “0x3a”、“0x74”为上传的 CRC16 数值，低字节在前，
   最后一字节 8A 为本指令添加的累加和校验。

其他命令：

0x03；0000，0011；

1：两个副载波，高速率；

0000：select = 0，address = 0；有处于选择状态的 vicc 应答；

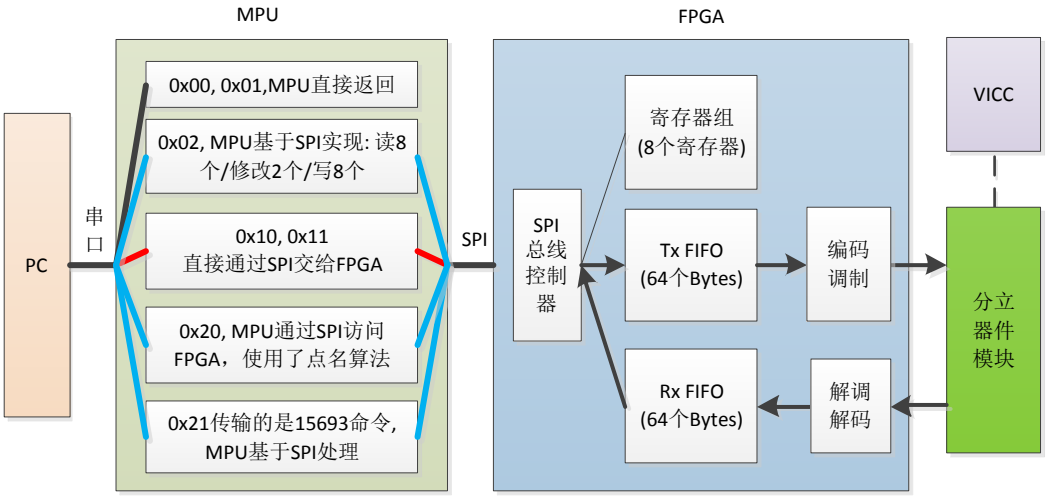
0x20：读单个块；

0x01：块号：0x01；

返回的 0x00，应该是 response.

但：上述内容，对于编码和调制实验有啥帮助呢？

通过上面解析，总结如下：



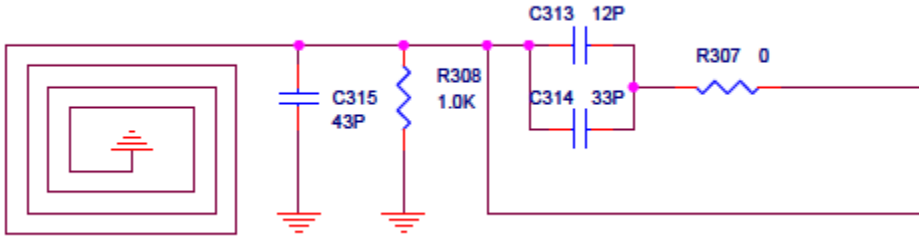
这么看：0x20,0x21 类命令，未必是直接写入 FPGA 的 FIFO 的。如果是的话，就可以通过 0x10，0x11 做了。因此 MPU 接收到 0x20，0x21 类命令后，重新翻译了一些访问寄存器，或者 FIFO 的命令？似乎也没有看到啊.

也不全对：比如收到 0x21 命令后，至少执行了写 FIFO，等待，读 FIFO 操作。

可能通过 0x10， 0x11 也可以实现。

1.2.4 原理图

试验中，测试点多次用 C315。C315 的位置即天线的输出位置。原理图中的位置如下：



1.2.5 实验内容

需要了解示波器：

靠某个点触发，获取当前的波形。

用的 inventory 命令。

1.2.5.1 载波

Part 3.3.2-3.3.3, page 87

直接测试；

1.2.5.2 ASK 调制

Part 3.3.5, page 90

使用软件：串口调试助手/或者 HFPRIN.exe

串口：输入：0x55 05 20 07 01 00 11 53 6c

分析：0x20：

0x07；多副载波；

0x01：inventory 命令，

0x00：mask len = 0;

按协议，通过天线发送的是：SOF + 0x07 + 0x01 + 0x00 + 0x11 + 0x53 + EOF

这如何观察到？

而且，需要先设置 FPGA 内部寄存器，说明使用 4 取 1 的 PPM 方式。

使用示波器的 CH1 触发采样，拿到的是 SOF。

1.2.5.3 FSK & PSK

Part 3.3.6/7, page 95

无内容；

1.2.5.4 副载波调制/解调

Part 3.3.8, page 96