文档编号：AN1047

上海东软载波微电子有限公司

应用笔记

HW2000B 应用注意事项

修订历史

版本	修订日期	修改概要
V1.0	2020-4-10	初版发布
V1.1	2020-7-16	增加状态切换说明（4.8 节） 增加低功耗说明（4.9 节） 版本从 V1.0 升级到 V1.1
V1.2	2021-5-31	增加可靠性初始化和异常复位的处理（4.10 节） 增加建议的过标配置（4.11 节） 更改初始化配置，增加频点和 Pipe 地址的初始化配置（5.5 节） 更改发射函数例程，增加 IRQ 中断方式例程和函数调用例程（5.7 节） 更改接收函数例程，将接收改为接收使能，接收处理，结束接收三个函数，增加 IRQ 中断方式例程和函数调用例程（5.8 节） 增加芯片从 Power Down 唤醒例程（5.14 节） 增加软复位例程（5.15 节） 增加发射功率配置例程（5.16 节） 版本从 V1.1 升级到 V1.2

地　址：中国上海市龙漕路 299 号天华信息科技园 2A 楼 5 层
邮　编：200235
E-mail：support@essemi.com
电　话：+86-21-60910333
传　真：+86-21-60914991
网　址：http://www.essemi.com

目 录

内容目录

图目录

表目录

第 1 章　　前言

本文针对用户在 HW2000B 芯片应用时可能碰到的问题进行说明，强调注意事项，协助用户尽快完成开发。文档整合了《AN1034_应用笔记_HW2000B_User_Guide》、《AN1035_应用笔记_HW2000B_Programming_Guide》两份应用笔记，补充注意事项和常见问题，增加的内容包括：

(1) 工作模式切换　　（2.1 小节）
(2) 数据包格式配置　　（2.3 小节）
(3) 状态切换说明　　（4.8 小节）
(4) 低功耗说明　　（4.9 小节）
(5) 模拟 SPI 的 C 语言驱动　　（5.4 小节）
(6) 灵敏度测试　　（6.1 小节）

第 2 章　工作模式

2.1　工作模式概述

HW2000B 芯片主要有 POWER DOWN、SLEEP、IDLE、TX、RX 五个工作模式，各个工作模式之间的切换见芯片数据手册的“芯片工作模式控制”章节内容。

需要注意的是，在工作模式之间切换时，必须结束上一个模式后，才能开启下一个模式。例如，把 HW2000B 从 TX 模式切换到 RX 模式，必须先关闭 TX 模式，延时一会，再打开 RX 模式，而不能直接配置寄存器从 TX 模式切换到 RX 模式。

2.2　链路模式简介

HW2000B 收发支持硬件链路控制（默认方式）与软件链路控制两种方式，可通过寄存器 PACK_LENGTH_EN (0x29[12])配置。

硬件链路控制方式又分为非定长模式和定长模式两种，默认为非定长模式。非定长模式支持两级收发 FIFO，每级 FIFO 最大支持 63 字节 PAYLOAD，FIFO 中的第一个字节的值确定所需读取 PAYLOAD 的长度，非定长模式支持 ACK 及 ACK PAYLOAD 功能。

定长模式配置可通过寄存器 0x31 配置，FIX_PLD_LEN_EN(0x31[7])为定长模式使能位，FIX_PLD_LEN(0x31[15:8])为定长模式 payload 长度配置寄存器。定长模式支持两级收发 FIFO，每级 FIFO 最大支持 63 字节 PAYLOAD。定长模式不支持 ACK 及 ACK PAYLOAD 功能。

软件链路控制方式只支持一级收发 FIFO，且不支持 ACK、ACK PAYLOAD、硬件 CRC 校验等功能。软件链路适用于 PAYLOAD 较长的的应用场合，PAYLOAD 长度由主控 MCU 控制。

2.3　数据包格式

HW2000B 不同的链路控制方式，数据包格式也有所不同，下面详细介绍各种链路方式下的数据包格式。

2.3.1　非定长模式数据包格式

如图 2-1 所示，为非定长模式数据包格式，每部分组成内容详见下文介绍。

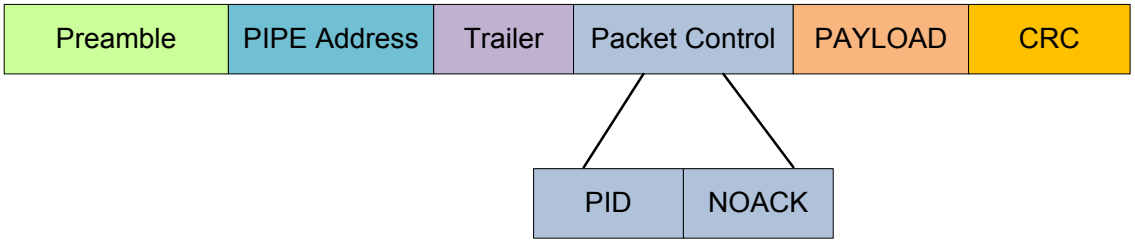

图 2-1 非定长模式数据包格式

Preamble

前导码支持 2,4,6…16bytes 长度可调，长度通过寄存器 PREAMBLE_LEN（PKTCTRL 寄存器的 Bit15-Bit13）配置。

PIPE Address (Syncword)

同步字支持 16/32/48bits 长度可调，长度通过寄存器 SYNCWORD_LEN（PKTCTRL 寄存器的 Bit12-Bit11）配置。支持 4 路数据通道，PIPE Address 可通过寄存器 0x40~0x47 配置。

Trailer

支持 4~18bits，长度可通过 TRAILER_LEN（PKTCTRL 寄存器的 Bit10-Bit8）配置。

PID

PID 长度为 2bits，发送时由硬件自动产生。PTX 每发送一次新的数据包 PID 将自动加‘1’。

NOACK

此指示位用于当 ACK 功能使能时，PTX 告知 PRX 当前包无需 ACK 的特例情况。可以通过 PTX_FIFOn_NOACK（FIFOnCTRL 寄存器的 Bit4）进行设置。

PAYLOAD

每级 FIFO 最大支持 63bytes 的 PAYLOAD，非定长模式 FIFO 的第一个 byte 代表 PAYLOAD 的长度，定长模式 PAYLOAD 长度由寄存器 FIX_PLD_LEN 配置。软件链路控制方式下，PAYLOAD 的长度由主控 MCU 芯片决定。

CRC

支持 CRC16 与 CRC8 两种模式，可通过 CRC_SEL（MISC1 寄存器的 Bit14）配置。

CRC16 生成多项式为：$x^{16}+x^{12}+x^5+1$。

CRC8 生成多项式为 ：x^8+x^2+x+1。

2.3.2 非定长模式ACK包格式

非定长模式下，若只使能 ACK 功能，不带 ACK PAYLOAD，PRX 返回的 ACK 包只包含 Preamble 和 PIPE Address 两部分，包格式如图 2-2 所示。

图 2-2 非定长模式 ACK 包格式

2.3.3 非定长模式ACK PAYLOAD包格式

非定长模式下，ACK PAYLOAD 数据包格式如图 2-3 所示。

图 2-3 非定长模式 ACK PAYLOAD 包格式

只有在非定长模式下，芯片才支持 ACK 带 PAYLOAD 功能，PRX 返回的 ACK 数据包包括

Preamble、 PIPE Address、Trailer、PAYLOAD 与 CRC，具体解释详见 2.3.1 包结构解释。

2.3.4 定长模式数据包格式

定长模式数据包格式如图 2-4 所示，与非定长模式数据包格式一致，唯一不同是在 PAYLOAD 部分，定长模式 PAYLOAD 就是用户希望传递的数据，数据长度由寄存器 FIX_PLD_LEN 配置，而非定长模式，PAYLOAD 的第一个字节是 PAYLOAD 的长度指示字节。

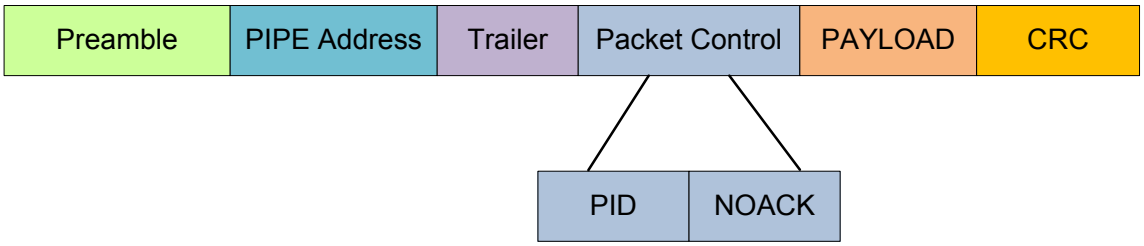

图 2-4 定长模式数据包格式

2.3.5 软件链路控制数据包格式

软件链路控制方式，数据包格式如图 2-5 所示。软件链路控制方式下不支持硬件 CRC 功能，PAYLOAD 的长度由主控 MCU 芯片决定。

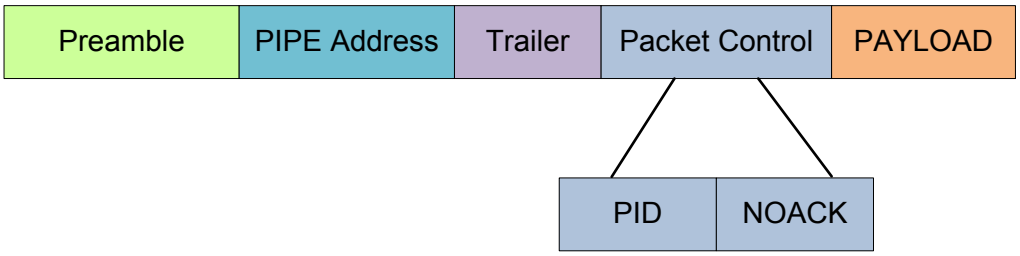

图 2-5 软件链路控制模式包格式

2.4 数据收发流程

HW2000B 支持硬件链路控制方式与软件链路控制方式，硬件链路控制方式又包括非定长模式跟定长模式两种，下面将介绍这几种模式的数据收发流程。

2.4.1 非定长模式

硬件链路控制方式需设置 PACK_LENGTH_EN 为‘1’。此模式支持两级收发 FIFO 与两级 ACKFIFO，每级 FIFO 最大支持 63bytes PAYLOAD，每级 ACKFIFO 最大支持 32bytes ACK PAYLOAD。

非定长模式下，PTX 在此控制方式下所填 FIFO/ACKFIFO 的第一个 byte 代表该级 FIFO/ACKFIFO 的 PAYLOAD 长度（填写值需大于‘0’）。

非定长模式下，PRX 可依据收取 PAYLOAD 的 FIFO 第一个 byte 值确定所需读取 PAYLOAD 的长度，类似的，PTX 可依据收取 ACK PAYLOAD 的 ACKFIFO 第一个 byte 值确定所需读取 ACK PAYLOAD 的长度。各级 FIFO 需满足条件才能进行收发。

2.4.1.1 PTX发送PAYLOAD流程

FIFO0 发送有效状态

- FIFO0_EN 为‘1’
- PTX_FIFO0_OCPY 为‘1’
- FIFO0 内 PAYLOAD 所属 PIPE 使能（例如使用 PIPE0 通信需使能 P0_EN 且设置 PTX_FIFO0_PIPE 为‘3’b000’）
- INT0 为‘0’

FIFO1 发送有效状态

- FIFO0 不处于发送有效状态（发送 FIFO0 优先级高于 FIFO1）
- FIFO1_EN 为‘1’
- PTX_FIFO1_OCPY 为‘1’
- FIFO1 内 PAYLOAD 所属 PIPE 使能（例如使用 PIPE0 通信，需使能 P0_EN 且设置 PTX_FIFO1_PIPE 为‘3’b000’）
- INT1 为‘0’

对于 PTX，通过 SPI 接口以不同的 FIFO 入口地址向 FIFO0(0x32)、FIFO1(0x33)填写发送的 PAYLOAD 数据。填写 PAYLOAD 完成之后，需软件填写 0x36/0x37 寄存器的 PTX_FIFOn_PIPE（默认为 PIPE0）与 PTX_FIFOn_OCPY 位（置‘1’），以指示所填的 FIFO 被占用并说明 FIFO 中 PAYLOAD 所属的 PIPE。

硬件状态机依次依据 FIFO0 与 FIFO1 的 PTX_FIFOn_OCPY 位并自动匹配 PTX_FIFOn_PIPE 位指示的 PIPE 地址，向 FIFO 中取数发送。

发送完成后状态机将检查是否有其它 FIFO 处于发送有效状态以继续发送。

待 PTX 发送完成中断 IRQ 置起后，需软件依次清相应 PTX_FIFOn_OCPY 位与中断标志位 INTn。（详见图 2-1）

2.4.1.2 PRX接收PAYLOAD流程

FIFO0 接收有效状态

- FIFO0_EN 为‘1’
- INT0 为‘0’

FIFO1 接收有效状态

- FIFO0 不处于接收有效状态（接收 FIFO0 优先级高于 FIFO1）
- FIFO1_EN 为‘1’
- INT1 为‘0’

对于 PRX，硬件在匹配接收 PIPE 地址之后，依次依据 FIFO 的有效状态将接收 PAYLOAD 填入处于接收有效状态下的 FIFO，填写完成之后自动将相应 PRX_FIFOn_OCPY 位置‘1’，并填写 PRX_FIFOn_PIPE 寄存器以指示该 FIFO 接收 PAYLOAD 所对应的 PIPE。

待接收 FIFO 中断 IRQ 置起后，软件需查询 INTn 指示位（0x3D）以确定收取 PAYLOAD 的

FIFO。读取 PAYLOAD 之后需将 INTn 标志清‘0’以保证后续 PAYLOAD 接收。

2.4.1.3 PRX发送ACK PAYLOAD流程

ACKFIFO0/1 满足发送条件

- ACKFIFO 配置寄存器 0x38/0x39 中所填 PIPE（PRX_ACKFIFOn_PIPE)与当前收取帧的 PIPE 地址匹配
- ACKFIFOn_EN 为‘1’
- PRX_ACKFIFOn_OCPY 为‘1’

当 PRX 使能 ACK 带 ACK PAYLOAD 功能时（Pn_EN = ‘1’，Pn_ACK_EN = ‘1’并且 Pn_ACKPAYLOAD_EN = ‘1’），PRX 在向 PTX 返回 ACK 的过程中自动从满足发送条件的 ACKFIFO 中取出 ACK PAYLOAD 进行发送。

2.4.1.4 PTX接收ACK PAYLOAD流程

ACKFIFO0 接收有效状态

- ACKFIFO0_EN 为‘1’
- ACKINT0 为‘0’

ACKFIFO1 接收有效状态

- ACKFIFO0 不处于接收有效状态（接收 ACKFIFO0 优先级高于 ACKFIFO1）
- ACKFIFO1_EN 为‘1’
- ACKINT1 为‘0’

当 PTX 使能 ACK 带 ACK PAYLOAD 功能时（Pn_EN = ‘1’，Pn_ACK_EN = ‘1’并且 Pn_ACKPAYLOAD_EN = ‘1’），PTX 在接收到 ACK PAYLOAD 后将数据填入满足条件的 ACKFIFO 中，填写完成之后自动将相应 PTX_ACKFIFOn_OCPY 位置‘1’，并填写 PTX_ACKFIFOn_PIPE 寄存器以指示该 ACKFIFO 接收 ACK PAYLOAD 所对应的 PIPE。

待 PTX 的中断置起后，软件需查询 ACKINTn 指示位（0x3D）以确定收取 ACK PAYLOAD 的 ACKFIFO。读取 ACK PAYLOAD 之后软件需将 ACKINTn 标志清‘0’以保证后续 ACK PAYLOAD 接收。

2.4.1.5 操作流程

- FIFO0 收发 ACK PAYLOAD 功能不使能

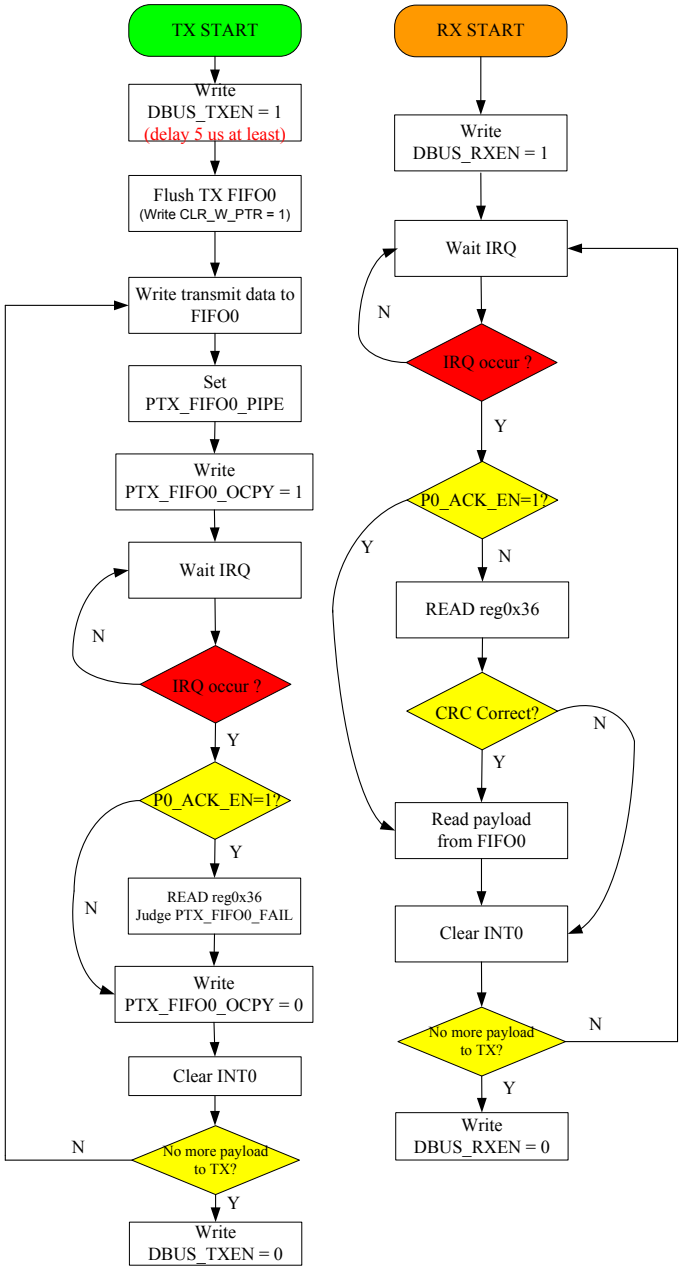

图 2-6 ACK PAYLOAD 功能不使能收发控制流程

上图所示为硬件链路收发方式下只使用 FIFO0 进行收发的操作流程，使用 FIFO1 或两级 FIFO 的流程基本类似，即需发送与接收 FIFO 满足上述条件才可进行正常收发流程。

- ✧ 若使能 ACK 功能（P0_ACK_EN = '1'），PTX 在读取中断 INT0 之后需检查 PTX_FIFO0_FAIL 标志位，以判断中断源为 PTX 接收 ACK 成功或是重传超时。
- ✧ PRX 在 ACK 不使能情况下，响应中断之后可以读取相应 FIFO 的 CRC 检测标志位判断接收 PAYLOAD 是否正确。
- ✧ PRX 在 ACK 使能情况下接收 PAYLOAD CRC 出错将自动重收，所以不会出现正常中断置起接收 PAYLOAD 出错的情况（无需检测 CRC）。

注 1：芯片上电默认为 SLEEP 状态，该状态下可进行寄存器初始化操作。

注 2：PTX 有发送需求使能 DBUS_TXEN 后，需等待 5us，系统时钟稳定后方可对 FIFO 进行操作。

◆ FIFO0 收发 ACK PAYLOAD 功能使能

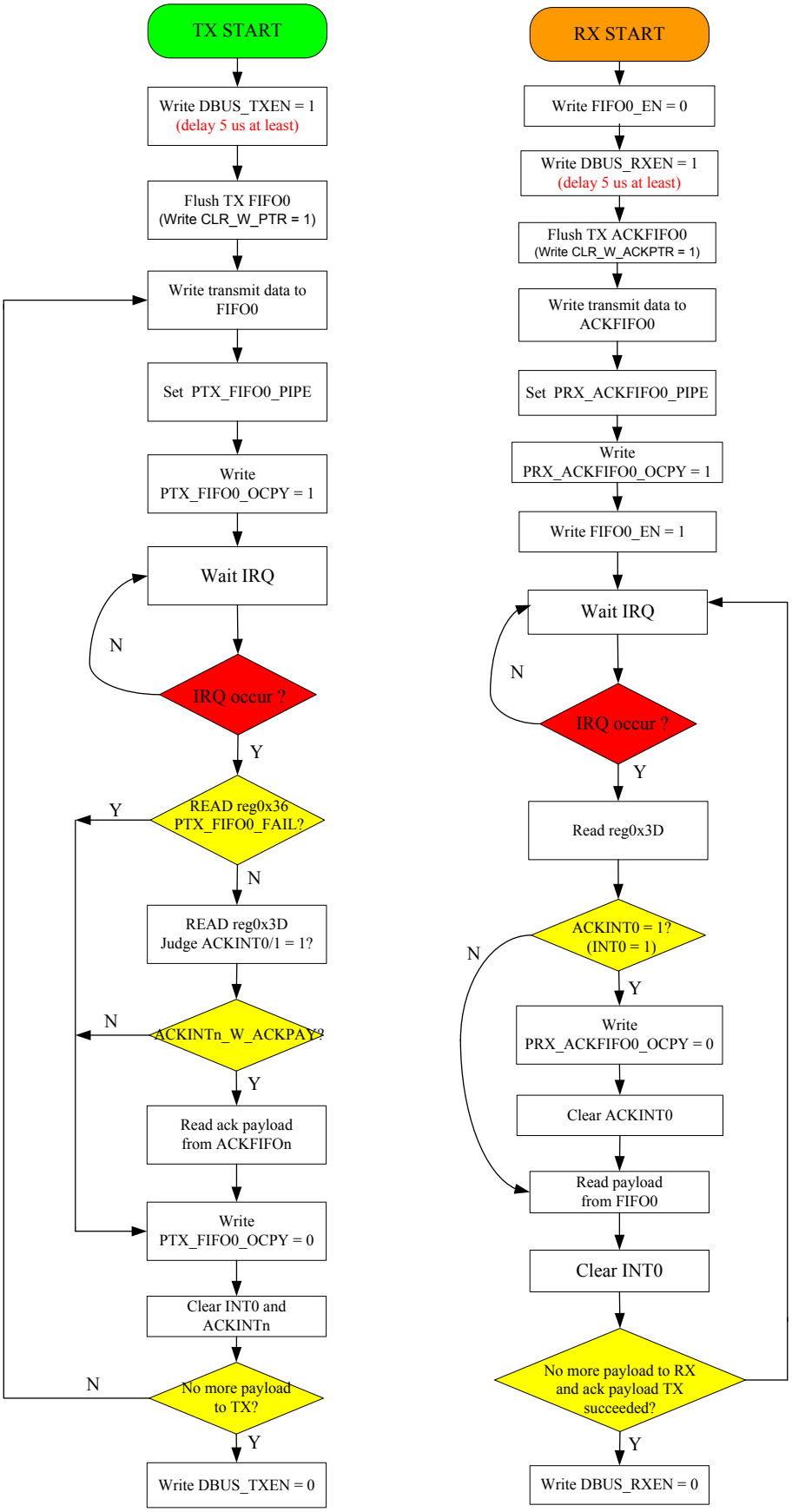

图 2-7 ACK 带 ACK PAYLOAD 收发控制流程

上图所示为只使用 FIFO0 收发 PAYLOAD、ACKFIFO0 发送 ACK PAYLOAD 的收发流程。

- ACKFIFO 的配置方式、流程与 FIFO 基本一致，PTX 需对发送 FIFO 与接收 ACKFIFO 进行配置，PRX 需对接收 FIFO 与发送 ACKFIFO 进行配置。
- PTX 在响应中断时需读取 0x3D 寄存器以确认哪个 ACKFIFO 成功接收 ACK PAYLOAD（优先级 ACKFIFO0 高于 ACKFIFO1）。
  例如 ACKINT0 ='1'，ACKINT0_W_ACKPAY ='1'表示 ACKFIFO0 成功接收 ACK PAYLOAD；ACKINT0 ='1'，ACKINT0_W_ACKPAY = '0'表示此次接收 ACK 过程中未发现 ACK PAYLOAD。
- PRX 在返回 ACK 过程中若无满足发送条件的 ACKFIFO，将只返回 ACK 信号，PTX 可依据 ACKINTn_W_ACKPAY 指示信号确认。
- PRX 在发送 ACK PAYLOAD 之后需在下一次收到同一 PIPE 新包之后才将 ACKINTn 标志置起。

◆ FIFO 数据包重复发送功能

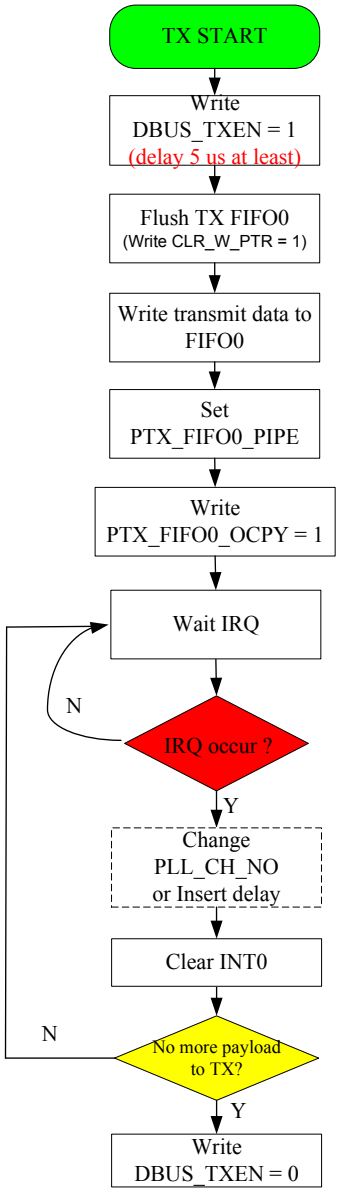

图 2-8 FIFO 数据包重复发送控制流程

芯片支持 FIFO 数据包重复发送功能，以满足某些数据包重复发送或快速跳频等应用场合，可以有效减少主控 MCU 的软件开销（如图 1-3 所示）。芯片在一次发送完成后只通过清中断标志 INT0 操作即可实现 FIFO 内数据包的重发流程，用户可在清中断标志 INT0 之前插入延时控制重发时间间隔或更改发送频点。

注：频点设置需在芯片发送或接收状态有效之前完成。

2.4.2 定长模式

定长模式下，基本流程同非定长模式一致，首先需设置 PACK_LENGTH_EN 为‘1’，即硬件链路控制方式。定长模式使能寄存器是 0x31 寄存器，首先配置 FIX_PLD_LEN_EN(0x31[7])为‘1’，使能定长模式，然后配置 FIX_PLD_LEN(0x31[15:8])寄存器，定义 payload 长度，长度最长为 63 Bytes。定长模式的 payload 第一字节不需要定义为长度指示字节，用户可自定义。

定长模式寄存器配置完成后，其发送流程、接收流程同非定长模式一样，同理 2.3.1 章节的收发流程。

在应用中，建议用户收发两端所使用的通信方式一样，即发射端若使用定长模式，则接收端同样使用定长模式，发射端使用非定长模式，接收端同样使用非定长模式。

在收、发两方都使用定长模式的情况下，0x31 寄存器的配置必须相同，且发射端写入 FIFO 的字节数与 FIX_PLD_LEN 定义的 payload 长度一致，接收端接收到数据后，读出 FIFO 的字节数同样也与 FIX_PLD_LEN 定义的 payload 长度相同。

若用户因方案需求，定长与非定长模式需混合使用，芯片只支持发射端用非定长模式，接收端用定长模式这一种组合；不支持发射端使用定长模式，接收端使用非定长模式这种组合。另外，需要说明的是，若发射端使用非定长模式，接收端使用定长模式，在配置接收端 FIX_PLD_LEN 寄存器时，长度需要比实际发射的 payload 长度少一个字节，但是实际读取 FIFO 时，按发射端实际发射 payload 长度读取，原因是，发射端使用非定长模式，首字节是长度指示字节，接收端使用定长模式，两种模式的处理不一致，没有长度指示字节，所以定长长度 FIX_PLD_LEN 需减 1。例如，发射端非定长模式需发射 5 个字节的数据，其第一字节是长度指示字节，必须填入 4，剩下 4 个字节用户可填入相关的用户数据；接收端使用定长模式，FIX_PLD_LEN 必须配置为 4，在接收中断产生后，需从 FIFO 中读取 5 个字节的 payload 数据。

2.4.3 软件链路控制方式

软件链路控制方式需设置 PACK_LENGTH_EN 为‘0’。此模式只支持 FIFO0 一级 FIFO，不支持 ACK 与硬件 CRC 校验功能。该模式下需要软件不断查询 FIFO0 的半空或半满标志位，完成对 FIFO0 的写入与读取操作，配合物理层硬件对 PAYLOAD 数据的发送与接收。具体操作流程可参考下图。

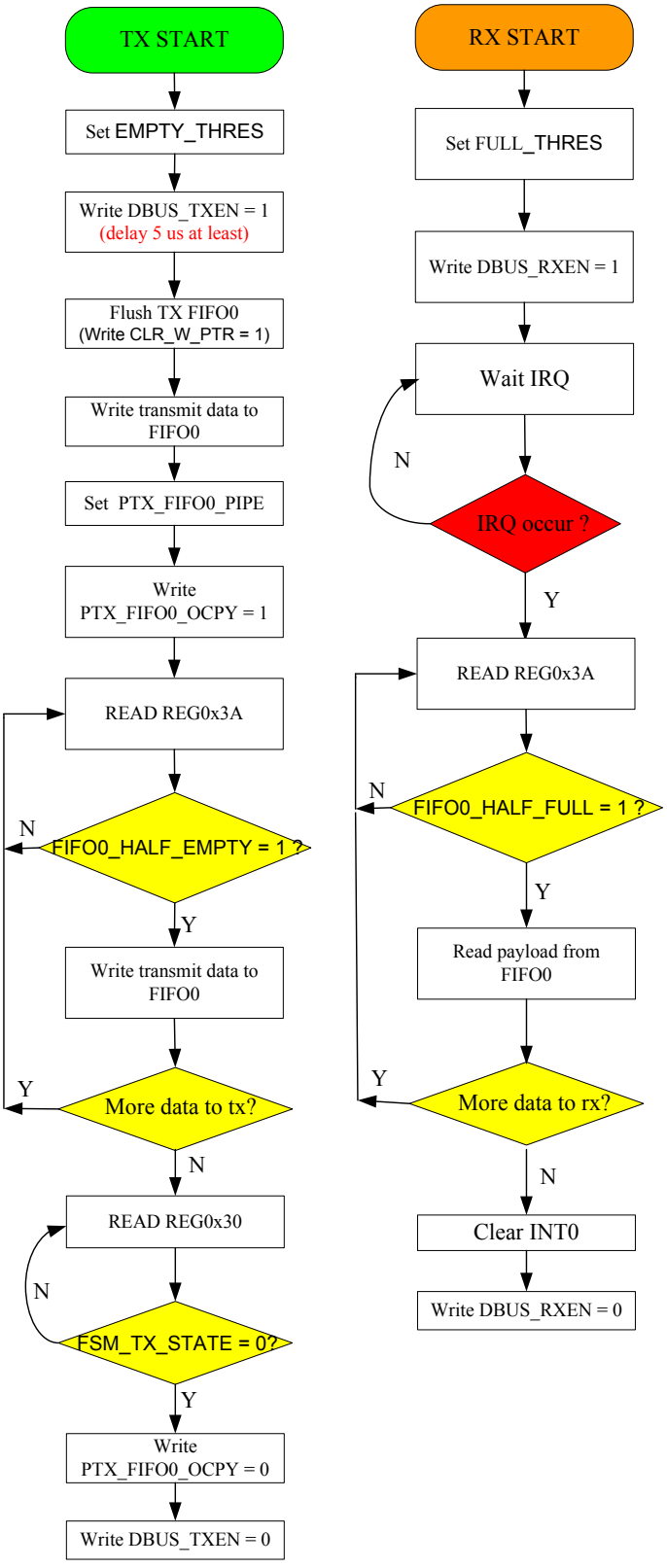

图 2-9 软件链路收发控制流程

注：可通过 EMPTY_THRES[4:0](0x28[15:11])，FULL_THRES[4:0](0x28[10:6])寄存器设置半空半满阈值，设置值需考虑 SPI 接口的访问速度。

第 3 章　寄存器配置

3.1　芯片初始化

- 芯片上电
- 寄存器初始化配置（以下配置不分先后）
  - 射频前端寄存器初始化，详见“寄存器初始化设置”章节。
  - 发送速率 RATE 设置(0x2A)，默认 1Mbps。
  - 外部晶振参考频率 REF_FQ，频点 PLL_CH_NO 设置(0x22)。
  - PREAMBLE 长度、SYNCWORD 长度、TRAILER 长度、编码方式、是否支持 FEC 设置(0x20)。
  - 同步字允许错误个数阈值 sync_thres 设置(0x28)。
  - 是否支持扰码功能，CRC8/16 选择设置(0x29)。
  - 使能通讯使用的 PIPE、设置 PIPE 地址、以及该 PIPE 是否支持 ACK(0x3C、0x40~0x47)。
  - 若支持 ACK 模式，设置重发次数 re_tx_times（0x23）。注意，对于 HW2000B，设置范围是 0~14，实际重发次数是设置值加 1，即 1~15 次。因此，设置值不能等于 15，否则会导致溢出而无法重发。需要指出的是，上一版本 HW2000 的设置范围是 1~15，其实际重发次数与设置值一致。

注：以上寄存器设置针对硬件链路控制方式，软件链路控制方式不支持 CRC、FEC、ACK 功能，可忽略相应寄存器配置步骤。

3.2　寄存器初始化设置

表 3-1 与表 3-2 分别为使用 12MHz 晶振时，1Mbps 与 250Kbps 的寄存器初始化配置(收发双方)，需按以下给定顺序进行初始化设置。

Reg address (Hex)	Default Value (Hex)	Recommend Value (12MHz crystal frequency) (Hex)
4C	0000	55AA（开启内部寄存器写使能）
50	0000	0000
51	0000	0000
52	0000	0000
53	0000	0001
54	0000	0002
55	0000	000A
56	0000	0012
57	0000	0212
58	0000	0412
59	0000	064A
5A	0000	084A

5B	0000	0A4A
5C	0000	0A52
5D	0000	0A92
5E	0000	0C92
5F	0000	0CD2
60	0000	0CDA
61	0000	0CE3
01	0000	4D58
02	0000	04CC
06	0000	B000
07	0000	54E0
08	0000	B6C4
09	0000	B442
0B	0860	0873
0F	0000	FC04
19	0000	2084
1B	0000	E754
1C	0000	51A0
20	5000	F000
26	000F	000C
27	8F0D	8F20
28	2103	8402
2A	C07E	C0FF
2C	8883	918B
48	4320	4300
49	1B30	1330
4A	0004	326C
4C	0000	FFFF（关闭内部寄存器写使能）

表 3-1 1Mbps 寄存器初始化设置

Reg address (Hex)	Default Value (Hex)	Recommend Value (12MHz crystal frequency) (Hex)
4C	0000	55AA（开启内部寄存器写使能）
50	0000	0000
51	0000	0000
52	0000	0000
53	0000	0001
54	0000	0002
55	0000	000A
56	0000	0012
57	0000	0212
58	0000	0412

59	0000	064A
5A	0000	084A
5B	0000	0A4A
5C	0000	0A52
5D	0000	0A92
5E	0000	0C92
5F	0000	0CD2
60	0000	0CDA
61	0000	0CE3
01	0000	4D58
02	0000	04CC
08	0000	B6C4
09	0000	B442
0B	0860	0873
0F	0000	FC04
19	0000	0884
1A	0000	0D31
20	5000	F000
26	000F	000C
27	8F0D	8F20
28	2103	8402
2A	C07E	40FF
2C	8883	918B
48	4320	4300
49	1B30	1330
4A	0004	326C
4C	0000	FFFF（关闭内部寄存器写使能）

表 3-2 250Kbps 寄存器初始化设置

芯片提供低功耗接收模式，在该模式下灵敏度下降约 6dB，接收功耗约为 18mA。需将表 3-1 与表 3-2 中 0x08 寄存器初始化值由 0xB6C4 改为 0x86C4。

3.3 晶振配置

若使用 16M 晶振，需另外添加如下寄存器初始化：

晶振（MHz）	寄存器配置(Hex)		
	0x04	0x1C	0x22
16	4800	5198	2030

表 3-3 16M 晶振参数配置

注：使用 16M 晶振，以上寄存器初始化需在表 3-1 或表 3-2 首尾寄存器初始化之间完成。

3.4 发射功率配置

图 3-1 为 HW2000B 芯片在 VDD = 3.3V 室温下的 PA 输出测试曲线图（横坐标为 0x0B[5:0]寄存器设置值，纵坐标为输出功率），表 3-4 为常用输出功率下的寄存器配置。

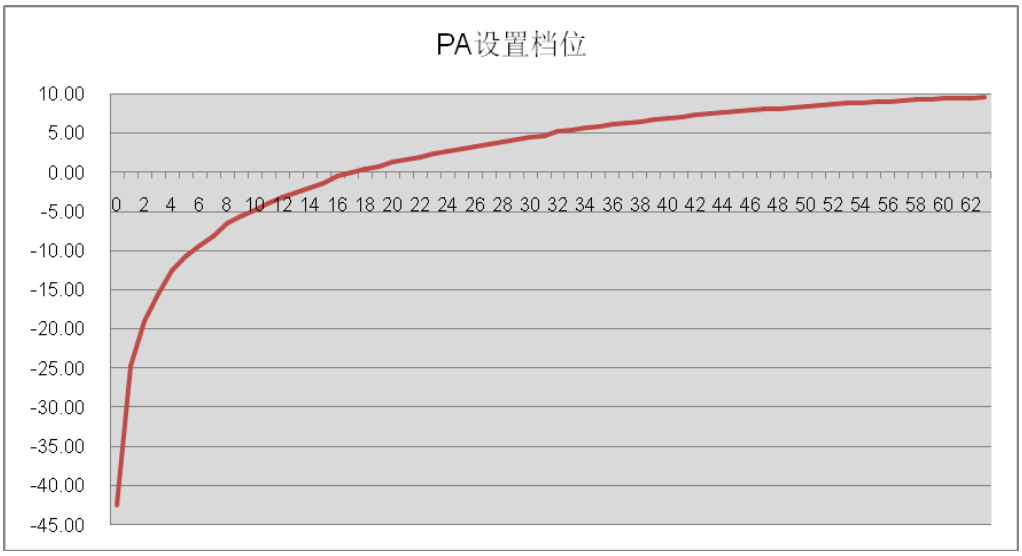

图 3-1 发送功率与配置

发送功率 (dBm)	寄存器配置(Hex)		
	0x0B	0x49	0x27
+9	0873	1330	8F20
+8	086C	1330	8F20
+6	0862	1330	8F20
+5	085F	1330	8F20
+4	085B	1330	8F20
+2	0855	1330	8F20
0	0851	1330	8F20
-5	084A	1330	8F20
-10	0845	1330	8F20
-15	0843	1330	8F20
-20	0842	1330	8F20
-25	0841	1330	8F20
-40	0840	1330	8F20

表 3-4 不同发送功率下寄存器配置值

注：以上配置仅作参考，芯片实际输出功率受 PCB 外围影响较大，芯片最终输出功率由 0x0B[5:0]设置值决定（设置值与输出功率趋势如图 2-1 所示），可在以上参考功率档位设置基础上调整 0x0B[5:0]设置值以得到实际需要的发送功率。

第 4 章　注意事项

4.1　RSSI读取

测试方法：

1. 芯片上电，初始化 HW2000B 寄存器，见“寄存器初始化设置”章节。
2. 将芯片设为 RX 状态。
3. 改变输入能量，接收端接收中断置起后，读取 pkg_rssi(0x2B，补码形式)锁存的 RSSI 值。若需检测环境能量，建议在 RX 接收使能后延时 350us 左右读取 RSSI(0x2D，补码形式)。

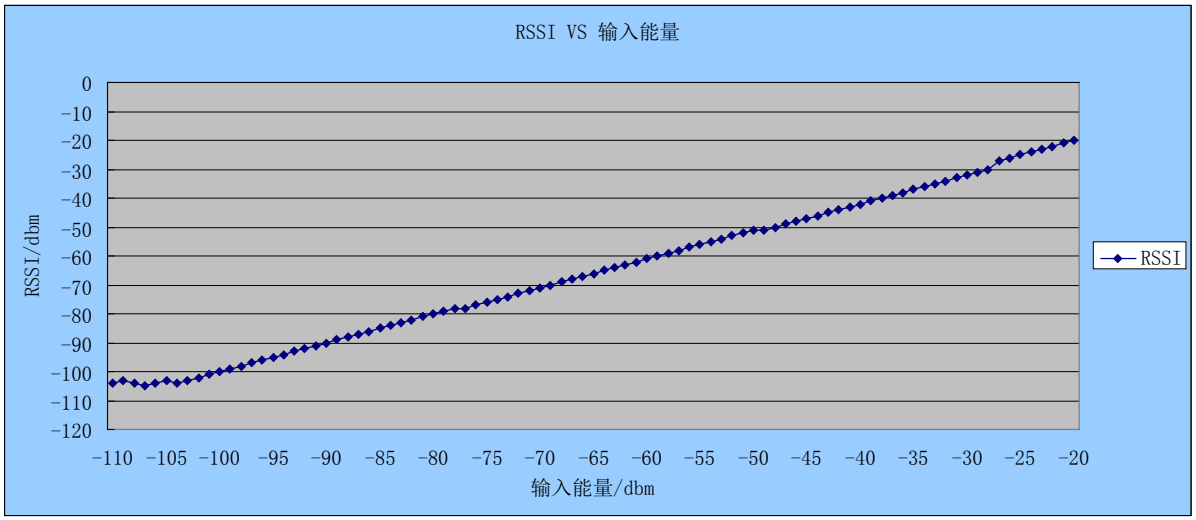

图 4-1 RSSI 寄存器与输入能量对应关系

4.2　CE、PD_CTRL和SFT_RST的区别

正常工作时，需将 CE 引脚拉高，使能 HW2000B 芯片。若拉低 CE 引脚将全局复位 HW2000B 芯片，包括复位芯片内部各状态信号与寄存器，拉高 CE 后建议延时 3ms 等待芯片稳定后，再进行寄存器初始化。

PD_CTR(0x23[15])为芯片进入 POWER DOWN 模式使能信号，仅控制芯片进入掉电模式，寄存器状态保持并可读写。

SFT_RST(0x23[14])为软件复位使能信号，仅复位芯片内部各状态信号，并不复位内部寄存器。也即 INT 中断信号等状态变量将会被复位， 同时 SFT_RST 使能后会将写保护关闭（0x4C 寄存器的值变为 0xFFFF），导致功率寄存器（0x0B）等未开放寄存器不能修改。

4.3　数据载荷说明

HW2000B 收发数据 FIFO 深度为 63，若 0x29[12]设置为‘1’，则 FIFO 第一个字节代表收发数据长度，用户数据需限制为 62 字节，注意 FIFO 第一字节所代表数据长度并未包含该字节本身。

4.4　状态查询模式说明

当通过 SPI 轮询方式检测 HW2000B 收发状态是否完成时，在轮询相关状态寄存器时需加入适

当延时，以避免 SPI 通信所产生的高次谐波可能对收发过程产生影响。
采用 IRQ 中断方式实现收发状态检测则无此问题产生。

4.5 POWER DOWN模式说明

HW2000B 进入 POWER DOWN 状态前，若芯片处于发射或接收使能的情况下，必须首先关闭发射或接收使能，然后再打开 POWER DOWN 使能信号 PD_CTRL。另外，在 POWER DOWN 状态下，MISO_TRI_OPT 位必须置 1，防止 MISO 管脚有漏电流出现，详见 5.13 节 POWER DOWN 参考例程。

4.6 ACK模式重发次数说明

HW2000B 使能 ACK 模式时，用户需配置最大重发次数，即在没有收到 ACK 的情况下，最大重发几次。重发次数配置寄存器为 0x23 的 RE_TX_TIMES【11:8】寄存器，可配置值为 0x0 ~ 0xE，而实际的最大重发次数是配置值加 1，比如配置值为 0，最大重发 1 次。注意不能配置为 0xF，原因是 0xF 加 1 后会导致数据溢出，不能重发。
另外，用户需要注意，HW2000 与 HW2000B 配置方式稍有改动，HW2000 的 ACK 重发次数可配置值的范围是 0x1 ~ 0xF，不能配置为 0，实际的最大重发次数就是配置值，比如，配置值为 3 时，最大重发次数就为 3。

4.7 ACK PAYLOAD模式说明

在 ACK PAYLOAD 功能使能模式下，因 ACKINTn(0x3D 寄存器)表征的是前一次接收数据时回复之 ACK PAYLOAD 是否成功，当接收到 PID 信号时对应 ACKINTn 产生，而 INTn 中断需要接收完整数据帧后才能产生。因此，在使能 ACK PAYLOAD 功能时，每次回复时需要同时使能 ACKFIFO0 和 ACKFIFO1，并写入相同返回数据，由状态机自动选取满足条件 ACKFIFO 发送数据。

4.8 状态切换说明

HW2000B 在收、发切换或收、发开关的情况下，为防止切换时间或开关时间不足导致状态机紊乱，建议在状态切换或开关中间加入软复位操作。具体如下：

- HW2000B 从 TX 状态转换到 RX 状态：首先关闭发射状态（包括清 PTX_FIFO0_OCPY，清 FIFO_INT 中断标志，关闭发射使能 DBUS_TXEN），然后添加 SFT_RST（0x23 寄存器）软复位操作，注意软复位使能后，需要再结束软复位操作，即软复位不使能（0x23 寄存器）。软复位操作后，打开接收使能，芯片进入接收状态。
- HW2000B 从 RX 状态转换到 TX 状态：首先关闭接收状态（包括清 FIFO_INT 中断标志，关闭接收使能 DBUS_RXEN），然后添加 SFT_RST（0x23）软复位操作，注意软复位使能后，需要再结束软复位操作，即软复位不使能（0x23 寄存器）。软复位操作后，打开发射使能，芯片进入发射状态。
- HW2000B TX 状态开关说明：若用户在使用时有发完一包数据后，立即发射下一包数据的用法，即发完一包数据后，关闭发射，中间没有任何延时操作，立即又打开发射使能，准备发射下一包数据，这种操作芯片内部状态机实际状态是 TX->IDLE->SLEEP->TX，内部状态机运行需要一定的时间，若用户 SPI 速率较高，可能发生状态机运行时间不够，导致状态机发生紊乱的可能，建议有以上用法的用户，在关闭 TX 使能后，添加 SFT_RST（0x60）软复位操作，注意软复位使能后，需要再结束软复位操作，即软复位不使能（0x23 寄存

器），然后再打开发射使能，发射下一包数据。

- HW2000B RX 状态开关说明：若用户在使用时有收完一包数据后，继续接收，等待收下一包数据的需求，一种方式是用户可以不关闭接收使能，只需在接收数据处理完后清 FIFO_INT 即可。第二种方式是接收完一包数据后，关闭接收使能，然后再打开接收，用户在操作中有可能出现关闭接收使能，又立即打开接收使能的情况，芯片内部状态机实际状态是 RX->IDLE->SLEEP->RX，内部状态机运行需要一定的时间，若用户 SPI 速率较高，可能发生状态机运行时间不够，导致状态机发生紊乱的可能，状态机紊乱就会出现接收死机的现象，建议有以上用法的用户，在关闭 RX 使能后，添加 SFT_RST（0x23 寄存器）软复位操作，注意软复位使能后，需要再结束软复位操作，即软复位不使能（0x23 寄存器），然后再打开 RX 使能，等待接收下一包数据

4.9 低功耗说明

对于低功耗的应用系统，HW2000B 不工作时可进入 POWER DOWN 状态。

HW2000B 与 MCU 的交互管脚主要包括：SPI 相关接口 CSN、SCK、MOSI、MISO；芯片使能管脚 CE；中断管脚 IRQ。

若这些管脚与 MCU 的 I/O 相连，只需在初始化时将各管脚进行初始化配置：CSN、SCK、MOSI 和 CE 在 MCU 端配置为数字输出管脚，CE 置高电平，CSN 在无 SPI 操作时置高电平；MISO 和 IRQ 在 MCU 端配置为数字输入管脚，并对 IRQ 进行端口中断配置。此外，若 MCU 具有 SPI 功能，可对 SPI 管脚（SCK、MOSI、MISO）进行 SPI 配置。

若上述初始化配置完成后，则对于 POWER DOWN 前的端口无需处理，直接调用 POWER DOWN 函数（5.13 小节），HW2000B 即可正常进入 POWER DOWN 状态。

若 IRQ 未和 MCU 的 I/O 相连，则 IRQ 管脚在 POWER DOWN 前无需处理。

若 CE 管脚直接接到电源，则无硬件复位功能，CE 管脚在 POWER DOWN 前无需处理。

用户可通过测量 VDD33A 和 VDD33D 两个电源管脚的电流来判断 HW2000B 是否进入 POWER DOWN 状态：两个电源管脚的电流和约 1.5uA，说明 HW2000B 已进入 POWER DOWN 状态。若用户系统休眠电流偏大，此方法可帮助用户快速判断是 MCU 还是 RF 漏电。

4.10 可靠性初始化和异常复位后的处理

对于干扰较大的应用环境，为了提高 HW2000B 的可靠性，可以按照以下两种方法进行处理。

1）开机不稳定的处理方法

对于一上电就不能通信的情况，可能是刚上电时的电源不稳定，导致 HW2000B 初始化不成功。这是因为，在进行初始化配置时，第一个初始化的寄存器 0x4C 是关闭写保护寄存器，若该寄存器未能正常写入 0x55AA，会导致后续被写保护的寄存器（如 0x01~0x19 等寄存器）写入失败。而寄存器初始化配置失败，会导致 HW2000B 不能正常通信。

其处理方法如下：

- 首先，对刚上电的 MCU 等主控芯片初始化后，需延时 50~100ms，再通过对 HW2000B 的 CE 脚拉低（即 CE=0，并延时 2ms）再拉高（即 CE=1，并延时 2ms）操作，进行芯片硬复位。

- 然后，对 HW2000B 进行初始化配置操作。需要注意的是，在初始化寄存器前，需通过调用 hw2000b_init_verify()函数，判断写保护寄存器 0x4C 是否被正常写入 0x55AA。该函数例程如下：

```
void   hw2000b_init_verify(void)
{
    uint16_t   reg;
    while(1){
        HW2000B_write_reg(0x4C, 0x55AA); //关闭写保护
        HW2000B_write_reg(0x54, 0x0002); //对被保护的寄存器进行写操作
        reg = HW2000B_read_reg(0x54);    //读被保护的寄存器，判断是否正常写入
        if(reg == 0x0002){     //写保护关闭，0x54 寄存器才能进行读/写
            break;
        }
    }
}
```

2）开机后工作一段时间不能通信的处理方法

对于开机后工作一段时间或者 EFT 测试时不能通信的现象，可能有以下两方面的原因：

- HW2000B 内部状态机发生紊乱，导致通信异常。

其处理方法是：在 HW2000B 每次进入发射或接收前进行软复位。可参考 5.7 或 5.8 小节的发射接收函数例程以及 5.15 小节的软复位例程。

- HW2000B 受到外部干扰而复位，导致通信异常。

其处理方法如下：

a）硬件处理

CE 管脚增加滤波电容，滤除引起芯片复位的外部干扰信号。该电容一般推荐取值为 10nF，若外部干扰较大，可根据需要换成 100nF 以上容值。

b）软件处理

◇ CE、IRQ 和 SPI 管脚配置为大电流驱动模式，增强管脚的抗干扰能力。

在 SPI 速率较快（>1Mbps）的情况下，配置如下：

```
HW2000B_write_reg(0x26, 0xFC0C);//大电流驱动打开
```

在 SPI 速率较慢（<1Mbps）的情况下，配置如下：

```
HW2000B_write_reg(0x26, 0xFC0F);//大电流驱动打开，SPI 内部滤波器打开
```

◇ 在系统资源允许的情况下，定时进行 HW2000B 的初始化配置，避免因芯片被外部干扰复位，导致通信异常。

◇ 若不能进行定时初始化，则需定时查询初始化配置过的寄存器（数据手册中可查询到的未写保护的寄存器）。若该寄存器被复位，说明 HW2000B 被复位，需重新进行初始化。例如，0x20 寄存器复位状态下的值是 0x5000，假设用户初始化时配置

为 0xF000，可定时去查询 0x20 寄存器的值，若变为复位值 0x5000，则需对 HW2000B 重新进行初始化。初始化的步骤仍是先通过 CE 脚进行硬复位，再对寄存器进行初始化配置。

4.11 推荐的过标初始化配置

若用户产品需要进行 CE、FCC 等标准认证，推荐使用如下初始化配置：

```
void HW2000B_init_250K(void)
{
    uint8_t i;
    uint16_t agcTab[18] = { 0x0000, 0x0000, 0x0000, 0x0001, 0x0002, 0x000A,
                            0x0012, 0x0212, 0x0412, 0x064A, 0x084A, 0x0A4A,
                            0x0A52, 0x0A92, 0x0C92, 0x0CD2, 0x0CDA, 0x0CE3
                           };

    HW2000B_write_reg(0x4C, 0x55AA);
    for (i = 0; i < 18; i++) {
        HW2000B_write_reg(0x50 + i, agcTab[i]);
    }
    HW2000B_write_reg(0x0F, 0XFC07);
    HW2000B_write_reg(0x01, 0x4D48);
    HW2000B_write_reg(0x02, 0x04CC);
    HW2000B_write_reg(0x08, 0xB6C4);
    HW2000B_write_reg(0x09, 0xBCC2);
    HW2000B_write_reg(0x26, 0x000C);
    HW2000B_write_reg(0x48, 0x4300);
    HW2000B_write_reg(0x4A, 0x326C);

    HW2000B_write_reg(0x0B, 0x0865);
    HW2000B_write_reg(0x49, 0x1330);
    HW2000B_write_reg(0x27, 0x8F20);

    HW2000B_write_reg(0x28, 0x8402);
    HW2000B_write_reg(0x2C, 0x918B);
    HW2000B_write_reg(0x2A, 0x40FF);
    HW2000B_write_reg(0x1A, 0x0D31);
    HW2000B_write_reg(0x19, 0x0884);
    HW2000B_write_reg(0x20, 0xF000);
    HW2000B_write_reg(0x05, 0x2051);
```

```
HW2000B_write_reg(0x0A, 0xD621);
```

第 5 章　参考例程

5.1　SPI帧格式和通讯时序

SPI 通讯帧格式和时序，可参考 HW2000B 芯片数据手册的“SPI 通信接口”章节。

5.2　SPI读写函数原型

通用 MCU 端需要根据 SPI 读写时序，实现寄存器读写和 FIFO 读写功能函数，函数原型如下：

```
/*****************************************************************************
* 函数名称：HW2000B_write_reg
* 功能描述：写 HW2000B 寄存器
* 输入参数：addr   寄存器地址
            value  寄存器值
* 返回参数：无
*****************************************************************************/
void HW2000B_write_reg(uint8_t addr, uint16_t value)

/*****************************************************************************
* 函数名称：HW2000B_read_reg
* 功能描述：读 HW2000B 寄存器
* 输入参数：addr   寄存器地址
* 返回参数：value  寄存器值
*****************************************************************************/
uint16_t HW2000B_read_reg(uint8_t addr)

/*****************************************************************************
* 函数名称：HW2000B_write_fifo
* 功能描述：写 HW2000B FIFO
* 输入参数：addr    FIFO 地址
            data    数据地址
            length  数据长度
*****************************************************************************/
void HW2000B_write_fifo(uint8_t addr, uint8_t *data, uint8_t length)
```

```
/*****************************************************************************
* 函数名称：HW2000B_read_fifo
* 功能描述：写 HW2000B FIFO
* 输入参数：addr    FIFO 地址
            data    数据地址
            length  数据长度
* 返回参数：无
*****************************************************************************/
void HW2000B_read_fifo(uint8_t addr, uint8_t *data, uint8_t length)
```

5.3 ES7P系列MCU SPI汇编驱动

用户若采用上海东软载波微电子有限公司 HR7P 系列或 ES7P 系列 MCU 作为主控，推荐采用以下 SPI 汇编驱动程序。

```
#define  PCSN    PB     //MCU 端口选择，用户需要初始化输入/输出属性
#define  PSCK    PA
#define  PMOSI   PB
#define  PMISO   PA

#define CSN      0x00   //MCU 端口比特位选择，用户需要初始化输入/输出属性
#define SCK      0x02
#define MOSI     0x01
#define MISO     0x03

void HW2000B_write_reg(uint8_t addr, uint16_t value)
{
    uint8_t i;

    __asm
    {
        MOVI    0x80
        IOR     (&addr&) % 0x80, 1

        BCC     PCSN, CSN

        CLR     (&i&)   % 0x80
```

```
JBC     (&addr&)   % 0x80, 7 ;write     addr
BSS     PMOSI, MOSI
JBS     (&addr&)   % 0x80, 7
BCC     PMOSI, MOSI
RL      (&addr&)   % 0x80
BSS     PSCK, SCK
INC     (&i&)   % 0x80, 1
BCC     PSCK, SCK
MOVI    0x08
XOR     (&i&)   % 0x80, 0
JBS     PSW, Z
GOTO    $-0x0B;

CLR     (&i&)   % 0x80

JBC     (&value&)   % 0x80 + 1, 7 ;write     value_h
BSS     PMOSI, MOSI
JBS     (&value&)   % 0x80 + 1, 7
BCC     PMOSI, MOSI
RL      (&value&)   % 0x80 + 1
BSS     PSCK, SCK
INC     (&i&)   % 0x80, 1
BCC     PSCK, SCK
MOVI    0x08
XOR     (&i&)   % 0x80, 0
JBS     PSW, Z
GOTO    $-0x0B

CLR     (&i&)   % 0x80

JBC     (&value&)   % 0x80, 7;write     value_l
BSS     PMOSI, MOSI
JBS     (&value&)   % 0x80, 7
BCC     PMOSI, MOSI
RL      (&value&)   % 0x80
BSS     PSCK, SCK
INC     (&i&)   % 0x80, 1
BCC     PSCK, SCK
```

```
            MOVI    0x08
            XOR     (&i&)   % 0x80, 0
            JBS     PSW, Z
            GOTO    $-0x0B

            BSS     PCSN, CSN
        }
    }

    uint16_t HW2000B_read_reg(uint8_t addr)
    {
        uint8_t i;
        uint16_t value;

        __asm
        {
            BCC     PCSN, CSN

            CLR     (&i&)   % 0x80

            JBC     (&addr&)   % 0x80, 7 ;write     addr
            BSS     PMOSI, MOSI
            JBS     (&addr&)   % 0x80, 7
            BCC     PMOSI, MOSI
            RL      (&addr&)   % 0x80
            BSS     PSCK, SCK
            INC     (&i&)   % 0x80, 1
            BCC     PSCK, SCK
            MOVI    0x08
            XOR     (&i&)   % 0x80, 0
            JBS     PSW, Z
            GOTO    $-0x0B;

            CLR     (&i&)   % 0x80

            BSS     PSCK, SCK                   ;read value
            BCC     PSW, C
            RL      (&value&)   % 0x80
```

```
            RL      (&value&)   % 0x80 + 1
            BCC     PSCK, SCK
            JBC     PMISO, MISO
            BSS     (&value&)   % 0x80, 0
            INC     (&i&)   % 0x80, 1
            MOVI    0x10
            XOR     (&i&)   % 0x80, 0
            JBS     PSW, Z
            GOTO    $-0x0B;

            BSS     PCSN, CSN
        }

        return value;
    }

    void HW2000B_write_fifo(uint8_t addr, uint8_t *data, uint8_t length)
    {
        uint8_t i, j;
        uint8_t   value;

        __asm
        {
            MOVI    0x80
            IOR     (&addr&) % 0x80, 1

            BCC     PCSN, CSN

            CLR     (&i&)   % 0x80

            JBC     (&addr&)   % 0x80, 7 ;write     addr
            BSS     PMOSI, MOSI
            JBS     (&addr&)   % 0x80, 7
            BCC     PMOSI, MOSI
            RL      (&addr&)   % 0x80
            BSS     PSCK, SCK
            INC     (&i&)   % 0x80, 1
            BCC     PSCK, SCK
```

```
            MOVI    0x08
            XOR     (&i&)   % 0x80, 0
            JBS     PSW, Z
            GOTO    $-0x0B;
        }

        for (j = 0; j < length; j++) {
            value = data[j];

            __asm
            {
                CLR     (&i&)   % 0x80

                JBC     (&value&)   % 0x80, 7;write data[j]
                BSS     PMOSI, MOSI
                JBS     (&value&)   % 0x80, 7
                BCC     PMOSI, MOSI
                RL      (&value&)   % 0x80
                BSS     PSCK, SCK
                INC     (&i&)   % 0x80, 1
                BCC     PSCK, SCK
                MOVI    0x08
                XOR     (&i&)   % 0x80, 0
                JBS     PSW, Z
                GOTO    $-0x0B
            }
        }

        __asm
        {
            BSS     PCSN, CSN
        }
    }

    void HW2000B_read_fifo(uint8_t addr, uint8_t *data, uint8_t length)
    {
        uint8_t i, j;
        uint8_t   value;
```

```
__asm
{
    BCC     PCSN, CSN

    CLR     (&i&)   % 0x80

    JBC     (&addr&)   % 0x80, 7 ;write     addr
    BSS     PMOSI, MOSI
    JBS     (&addr&)   % 0x80, 7
    BCC     PMOSI, MOSI
    RL      (&addr&)   % 0x80
    BSS     PSCK, SCK
    INC     (&i&)   % 0x80, 1
    BCC     PSCK,    SCK
    MOVI    0x08
    XOR     (&i&)   % 0x80, 0
    JBS     PSW, Z
    GOTO    $-0x0B;
}

for (j = 0; j < length; j++) {
    __asm
    {
        CLR     (&i&)   % 0x80

        BSS     PSCK, SCK                   ;read data
        BCC     PSW, C
        RL      (&value&)   % 0x80
        BCC     PSCK, SCK
        JBC     PMISO, MISO
        BSS     (&value&)   % 0x80, 0
        INC     (&i&)   % 0x80, 1
        MOVI    0x08
        XOR     (&i&)   % 0x80, 0
        JBS     PSW, Z
        GOTO    $-0x0A;
    }
```

```
            data[j] = value;
        }

        __asm
        {
            BSS       PCSN, CSN
        }
    }
```

5.4 SPI模拟C语言驱动

用户若采用模拟 SPI，通用的 C 语言驱动函数如下。

```
#define CSN         PB4
#define SCK         PB5
#define MOSI        PB6
#define MISO        PB7

void hw2000B_write_reg(uint8_t addr, uint16_t value)
{
    uint8_t i;
    addr |= 0x80;

    CSN = 0;
    for (i = 0; i < 8; i++) {
        if (addr & 0x80) {
            MOSI = 1;
        }
        else {
            MOSI = 0;
        }
        SCK = 1;
        addr <<= 1;
        SCK = 0;
    }

    for (i = 0; i < 16; i++) {
        if (value & 0x8000) {
            MOSI = 1;
        }
```

```
            else {
                MOSI = 0;
            }
            SCK = 1;
            value <<= 1;
            SCK = 0;
        }
        CSN = 1;
        MOSI = 0;
    }

    uint16_t hw2000B_read_reg(uint8_t addr)
    {
        uint8_t i;
        uint16_t value;

        CSN = 0;
        for (i = 0; i < 8; i++) {
            if (addr & 0x80) {
                MOSI = 1;
            }
            else {
                MOSI = 0;
            }
            SCK = 1;
            addr <<= 1;
            SCK = 0;
        }

        for (i = 0; i < 16; i++) {
            SCK = 1;
            value <<= 1;
            SCK = 0;
            if (MISO) {
                value |= 0x0001;
            }
        }
        CSN = 1;
```

```
        MOSI = 0;

        return value;
    }

    void hw2000B_write_fifo(uint8_t addr, uint8_t *data, uint8_t length)
    {
        uint8_t i, j;
        uint8_t value;
        addr |= 0x80;

        CSN = 0;
        for (i = 0; i < 8; i++) {
            if (addr & 0x80) {
                MOSI = 1;
            }
            else {
                MOSI = 0;
            }
            SCK = 1;
            addr <<= 1;
            SCK = 0;
        }

        for (j = 0; j < length; j++) {
            value = data[j];
            for (i = 0; i < 8; i++) {
                if (value & 0x80) {
                    MOSI = 1;
                }
                else {
                    MOSI = 0;
                }
                SCK = 1;
                value <<= 1;
                SCK = 0;
            }
        }
```

```
    CSN = 1;
    MOSI = 0;
}

void hw2000B_read_fifo(uint8_t addr, uint8_t *data, uint8_t length)
{
    uint8_t i, j;
    uint8_t value;

    CSN = 0;
    for (i = 0; i < 8; i++) {
        if (addr & 0x80) {
            MOSI = 1;
        }
        else {
            MOSI = 0;
        }
        SCK = 1;
        addr <<= 1;
        SCK = 0;
    }

    for (j = 0; j < length; j++) {
        for (i = 0; i < 8; i++) {
            SCK = 1;
            value <<= 1;
            SCK = 0;
            if (MISO) {
                value |= 0x0001;
            }
        }
        data[j] = value;
    }
    CSN = 1;
    MOSI = 0;
}
```

5.5 HW2000B 默认寄存器初始化

HW2000B 上电后需要初始化以下寄存器，1Mbps 和 250Kbps 初始化函数如下。

```
void HW2000B_init_250K(void)
{
    uint8_t i;
    uint16_t agcTab[18] = { 0x0000, 0x0000, 0x0000, 0x0001, 0x0002, 0x000A,
                            0x0012, 0x0212, 0x0412, 0x064A, 0x084A, 0x0A4A,
                            0x0A52, 0x0A92, 0x0C92, 0x0CD2, 0x0CDA, 0x0CE3
                           };

    HW2000B_write_reg(0x4C, 0x55AA);
    for (i = 0; i < 18; i++) {
        HW2000B_write_reg(0x50 + i, agcTab[i]);
    }
    HW2000B_write_reg(0x0F, 0XFC04);
    HW2000B_write_reg(0x01, 0x4D58);
    HW2000B_write_reg(0x02, 0x04CC);
    HW2000B_write_reg(0x08, 0xB6C4);
    HW2000B_write_reg(0x09, 0xB442);
    HW2000B_write_reg(0x4A, 0x326C);
    HW2000B_write_reg(0x0B, 0x0873);
    HW2000B_write_reg(0x49, 0x1330);
    HW2000B_write_reg(0x27, 0x8F20);
    HW2000B_write_reg(0x48, 0x4300);
    HW2000B_write_reg(0x26, 0x000C);
    HW2000B_write_reg(0x28, 0x8402);
    HW2000B_write_reg(0x2C, 0x918B);
    HW2000B_write_reg(0x2A, 0x40FF);
    HW2000B_write_reg(0x1A, 0x0D31);
    HW2000B_write_reg(0x19, 0x0884);
    HW2000B_write_reg(0x20, 0xF000);
    HW2000B_write_reg(0x22, 0x1830); //默认 2.450GHz 频点配置，建议用户更改频点
    HW2000B_write_reg(0x40, 0xE7E7); //默认 pipe 地址，建议用户更改为自定义地址
    HW2000B_write_reg(0x41, 0xE7E7); //默认 pipe 地址，建议用户更改为自定义地址
    HW2000B_write_reg(0x42, 0xE7E7); //默认 pipe 地址，建议用户更改为自定义地址
}
```

```
void HW2000B_init_1M(void)
{
    uint8_t i;
    uint16_t agcTab[18] = { 0x0000, 0x0000, 0x0000, 0x0001, 0x0002, 0x000A,
                            0x0012, 0x0212, 0x0412, 0x064A, 0x084A, 0x0A4A,
                            0x0A52, 0x0A92, 0x0C92, 0x0CD2, 0x0CDA, 0x0CE3
                            };
    HW2000B_write_reg(0x4C, 0x55AA);
    for (i = 0; i < 18; i++) {
        HW2000B_write_reg(0x50 + i, agcTab[i]);
    }
    HW2000B_write_reg(0x0F, 0xFC04);
    HW2000B_write_reg(0x01, 0x4D58);
    HW2000B_write_reg(0x02, 0x04CC);
    HW2000B_write_reg(0x08, 0xB6C4);
    HW2000B_write_reg(0x09, 0xB442);
    HW2000B_write_reg(0x4A, 0x326C);
    HW2000B_write_reg(0x0B, 0x0873);
    HW2000B_write_reg(0x49, 0x1330);
    HW2000B_write_reg(0x27, 0x8F20);
    HW2000B_write_reg(0x48, 0x4300);
    HW2000B_write_reg(0x26, 0x000C);
    HW2000B_write_reg(0x28, 0x8402);
    HW2000B_write_reg(0x2C, 0x918B);
    HW2000B_write_reg(0x1B, 0xE754);
    HW2000B_write_reg(0x06, 0xB000);
    HW2000B_write_reg(0x07, 0x54E0);
    HW2000B_write_reg(0x1C, 0x51A0);
    HW2000B_write_reg(0x19, 0x2084);
    HW2000B_write_reg(0x20, 0xF000);
    HW2000B_write_reg(0x2A, 0xC0FF);
    HW2000B_write_reg(0x22, 0x1830); //默认 2.450GHz 频点配置，建议用户更改频点
    HW2000B_write_reg(0x40, 0xE7E7); //默认 pipe 地址，建议用户更改为自定义地址
    HW2000B_write_reg(0x41, 0xE7E7); //默认 pipe 地址，建议用户更改为自定义地址
    HW2000B_write_reg(0x42, 0xE7E7); //默认 pipe 地址，建议用户更改为自定义地址
}
```

5.6 HW2000B TX CW

HW2000B 提供单载波发送模式，可用于频点和发送功率测试。

```
void power_test(void) //使能单载波发送
{
    _reg = HW2000B_read_reg(0x1C); //读出寄存器 0x1C 的原始值，关闭单载波发送时，
                                   //0x1C 寄存器的值需还原为原始值
    HW2000B_write_reg(0x1C, _reg&0xFE7F);
    HW2000B_write_reg(0x29, 0x0000);
    HW2000B_write_reg(0x21, 0x0100);
    HW2000B_write_reg(0x36, 0x0081);
}

void power_test_cancel(void) //关闭单载波发送
{
    HW2000B_write_reg(0x36, 0x0080);
    HW2000B_write_reg(0x21, 0x0000);
    HW2000B_write_reg(0x29, 0x1800);
    HW2000B_write_reg(0x3D, 0x0008);
    HW2000B_write_reg(0x1C,  _reg); //还原 0x1C 寄存器的原始值
}
```

5.7 HW2000B TX NOACK

以下配置可实现 HW2000B 最基本的 FIFO 发射功能，ACK 功能不使能。用户需参考 SPI 时序实现相关 SPI 操作函数。

（1）发射函数原型 1（查询寄存器方式）:

```
void hw2000b_tx_data(uint8_t *data)
{
    uint16_t   i = 30;
    hw2000b_sft_rst()；//发射使能前，先软复位（函数原型参考 5.15），复位芯片状态机
    HW2000B_write_reg(0x21, 0x0100);      // TX enable
    delay_us(5); //延时>=5us
    HW2000B_write_reg(0x3B, 0x8000); //清发射 FIFO
    HW2000B_write_fifo(0x32, data, _data[0]+1);
    HW2000B_write_reg(0x36, 0x0081);
    HW2000B_write_reg(0x37, 0x0000);
```

```
    delay_ms(2); //延时 2ms
    _reg = HW2000B_read_reg(0x3D);    //发射状态判断，也可由 IRQ 中断实现
    while (!(_reg & 0x0001)) {
        delay_ms(2); //延时 2ms
        __reg = HW2000B_read_reg(0x3D);
        i--;
        if(i<2){
            hw2000b_sft_rst()；//发射超时，软复位，复位芯片状态机
            break;
        }
    }

    HW2000B_write_reg(0x36, 0x0080); // FIFO occupy disable
    HW2000B_write_reg(0x3D, 0x0008); // clear INT0
    HW2000B_write_reg(0x21, 0x0000); // TX disable
}
```

（2）发射函数原型 2（IRQ 中断方式）：

```
void hw2000b_tx_data(uint8_t *data)
{
    uint16_t   i = 30000;

    hw2000b_sft_rst()；//发射使能前，先软复位（函数原型参考 5.15），复位芯片状态机
    _hw2000b_irq_request = 0;
    HW2000B_write_reg(0x21, 0x0100);      // TX enable
    delay_us(5); //延时>=5us
    HW2000B_write_reg(0x3B, 0x8000); //清发射 FIFO
    HW2000B_write_fifo(0x32, data, _data[0]+1);
    HW2000B_write_reg(0x36, 0x0081);
    HW2000B_write_reg(0x37, 0x0000);

    while (!_hw2000b_irq_request) {      //等待发射完成标志信号置起
        i--;
        if(i<2){
            hw2000b_sft_rst()；//发射超时，软复位，复位芯片状态机
            break;
        }
    }
```

```
    HW2000B_write_reg(0x36, 0x0080); // FIFO occupy disable
    HW2000B_write_reg(0x3D, 0x0008); // clear INT0
    HW2000B_write_reg(0x21, 0x0000); // TX disable
}
```

函数调用示例：

```
_data[0] = 15;    //数据格式_data[]={15, 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
for (i = 1; i < 16; i++){
    _data[i] = i;
}
hw2000b_tx_data(_data);
```

5.8 HW2000B RX NOACK

以下配置可实现 HW2000B 最基本的 FIFO 接收功能，ACK 功能不使能。

（1）接收使能函数

```
void hw2000b_rx_enable(void)
{
    hw2000b_sft_rst()；//接收使能前，先软复位（函数原型参考 5.15），复位芯片状态机

    HW2000B_write_reg(0x3D, 0x0008);   //clear INT0
    HW2000B_write_reg(0x21, 0x0000);   //RX disable

    HW2000B_write_reg(0x36, 0x0080);      //接收 FIFO 使能
    HW2000B_write_reg(0x37, 0x0000);
    HW2000B_write_reg(0x21, 0x0080);   //RX enable
}
```

（2）关闭接收函数

```
void hw2000b_rx_disable(void)
{
    HW2000B_write_reg(0x3D, 0x0008);   //clear INT0
    HW2000B_write_reg(0x21, 0x0000);   //RX disable

    hw2000b_sft_rst()；//接收使能前，先软复位（函数原型参考 5.15），复位芯片状态机
}
```

（3）接收处理函数 1(查询寄存器方式)

```
int8_t hw2000b_rx_data(uint8_t *data)
{
```

```
    uint16_t   reg;
    reg = HW2000B_read_reg(0x3D);    //接收状态判断，也可通过 IRQ 中断实现
    if(_(_reg & 0x0001) == 0x0001 ){
        _hw2000b_irq_request = 0 ;
        reg = HW2000B_read_reg(0x36);
        if((reg & 0x2000)==0){
            HW2000B_read_fifo(0x32, data, 1);
            HW2000B_read_fifo(0x32, &data[1], data[0]);
            HW2000B_write_reg(0x3D, 0x0008);   //clear INT0
            return   0;
        } else {
            HW2000B_write_reg(0x3D, 0x0008);   //clear INT0
            return   -1;
        }
    } else {
        return   -1;
    }
}
```

（4）接收处理函数 2(IRQ 中断方式)

```
int8_t hw2000b_rx_data(uint8_t *data)
{
    uint16_t   reg;
    if(_hw2000b_irq_request){ //判断是否收到数据，IRQ 中断是否置起
        _hw2000b_irq_request = 0 ;
        reg = HW2000B_read_reg(0x36);    //CRC 校验
        if((reg & 0x2000)==0){
            HW2000B_read_fifo(0x32, data, 1);
            HW2000B_read_fifo(0x32, &data[1], data[0]);
            HW2000B_write_reg(0x3D, 0x0008);   //clear INT0
            return   0;
        } else {
            HW2000B_write_reg(0x3D, 0x0008);   //clear INT0
            return   -1;
        }
    } else {
        return   -1;
    }
}
```

函数调用示例：

```
//需要接收时，首先配置接收使能函数
hw2000b_rx_enable()；

//接收使能后，系统可以调用接收函数，判断是否接收到数据
if(hw2000b_rx_data(_data) == 0){
    //接收到数据，进行处理
}

//接收处理结束，不需要接收时，可以调用接收关闭函数
hw2000b_rx_disable();
```

5.9 HW2000B TX ACK NOPAYLOAD

以下配置可实现 HW2000B 的 TX ACK 接收功能，且 ACK PAYLOAD 不使能。

```
HW2000B_write_reg(0x23, 0x0380 ); // ack times = 3
HW2000B_write_reg(0x3C, 0x1000 | 0x0001); // pipe 0 ack enable
while (1) {
    for (i = 1; i < 16; i++) {     //TX example data
        _txdata[i] = i;
    }
    _txdata[0] = 15;

    HW2000B_write_reg(0x21, 0x0100);// TX enable
    delay_us(5); //延时>=5us
    HW2000B_write_reg(0x3B, 0x8000); // FIFO write pointer clear
    HW2000B_write_fifo(0x32, _txdata, _txdata[0]+1); // write data
    HW2000B_write_reg(0x36, 0x0081); // fifo0, pipe0 enable , fifo0 ocupy
    //若 0x36 写入 0x0091 表示当前发送数据不需要接收端 ack 回复
    //HW2000B_write_reg(0x36, 0x0091); // fifo0, pipe0 enable , fifo0 ocupy, no ack
    HW2000B_write_reg(0x37, 0x0000);

    delay_ms(5); //延时 5ms
    _reg = HW2000B_read_reg(0x3D);  // waiting for tx finish，也可用 IRQ 中断实现
    while (!(_reg & 0x0001)) {
        delay_ms(5); //延时 5ms
        _reg = HW2000B_read_reg(0x3D);
    }
```

```
    _reg = HW2000B_read_reg(0x36);
    if (_reg & 0x8000) {
        // 重发超时，发射端未收到 ack 回复，认为发送失败
     }

    HW2000B_write_reg(0x36, 0x0080); // disable all
    HW2000B_write_reg(0x3D, 0x0008); // clear INT0
    HW2000B_write_reg(0x21, 0x0000);// TX disable
}
```

5.10 HW2000B RX ACK NOPAYLOAD

以下配置可实现 HW2000B RX ACK 发射功能，且 ACK PAYLOAD 不使能。

```
HW2000B_write_reg(0x23, 0x0380 ); // ack times = 3
HW2000B_write_reg(0x3C, 0x1000 | 0x0001); // pipe 0 ack enable
while (1) {
    for (i = 0; i < 64; i++) { //clear rx buffer
        _rxdata[i] = 0;
    }

    HW2000B_write_reg(0x36, 0x0080); //FIFO0 config
    HW2000B_write_reg(0x37, 0x0000); //FIFO1 bypass
    HW2000B_write_reg(0x21, 0x0080); //RX enable

    _reg = HW2000B_read_reg(0x3D);  //Waiting for rx finish，也可用 IRQ 中断实现
    while (!(_reg & 0x0001)) {
        delay_ms(5); //延时 5ms
        _reg = HW2000B_read_reg(0x3D);
    }
    _reg = HW2000B_read_reg(0x36);
    if (!(_reg & 0x2000)) {  //CRC 校验
        HW2000B_read_fifo(0x32, _data, 1);
         HW2000B_read_fifo(0x32, &_data[1], _data[0]);
    }

    HW2000B_write_reg(0x3D, 0x0008); //Clear INT0
    HW2000B_write_reg(0x21, 0x0000); //RX disable
}
```

5.11 HW2000B TX ACK PAYLOAD

以下配置可实现 HW2000B TX ACK 接收功能，并使能 ACK PAYLOAD。

```
HW2000B_write_reg(0x23, 0x0380 ); // ack times = 3
HW2000B_write_reg(0x3C, 0x1000 | 0x0011); // pipe 0 ack, ackpayload enable

while (1) {
    for (i = 1; i < 16; i++) { //tx   example data
        _txdata[i] = i;
    }
    _txdata[0] = 15;

    for (i = 0; i < 32; i++) { // clear rxack buffer
        _rxackdata[i] = 0;
    }

    HW2000B_write_reg(0x21, 0x0100);// TX enable
    delay_us(5); //延时>=5us
    HW2000B_write_reg(0x3B, 0x8000); // FIFO write pointer clear
    HW2000B_write_fifo(0x32, _txdata, _txdata[0]+1); // write data

    HW2000B_write_reg(0x36, 0x0081); // fifo0, pipe0 enable , fifo0 ocupy
    //若 0x36 写入 0x0091 表示当前发送数据不需要接收端 ack 回复
    //HW2000B_write_reg(0x36, 0x0091); // fifo0, pipe0 enable , fifo0 ocupy, no ack
    HW2000B_write_reg(0x37, 0x0000);
    HW2000B_write_reg(0x38, 0x0080); // ACKFIFO0 config
    HW2000B_write_reg(0x39, 0x0000); // ACKFIFO1 disable

    delay_ms(5); //延时 5ms
    _reg = HW2000B_read_reg(0x3D);// waiting for tx finish，也可用 IRQ 中断方式实现
   while (!(_reg & 0x0001)) {
        delay_ms(5); //延时 5ms
        _reg = HW2000B_read_reg(0x3D);
    }

    if (HW2000B_read_reg(0x36) & 0x8000) {
        //重发超时异常，用户自行处理
    } else if ((_reg & 0x0102) == 0x0102) { // ackint0 and ack with payload
```

```
            HW2000B_read_fifo(0x34, _rxackdata, 1);
            HW2000B_read_fifo(0x34, &_rxackdata[1], _rxackdata[0]);
            //ACK payload  接收数据
        }

        HW2000B_write_reg(0x36, 0x0080); // disable all
        HW2000B_write_reg(0x3D, 0x0808); // clear INT0 and ACKINT0
        HW2000B_write_reg(0x21, 0x0000);// TX disable
    }
```

5.12 HW2000B RX ACK PAYLOAD

以下配置可实现 HW2000B RX ACK 发射功能，并使能 ACK PAYLOAD。

```
    HW2000B_write_reg(0x23, 0x0380); // ack times = 3
    HW2000B_write_reg(0x3C, 0x1000 | 0x0011); // pipe 0 ack, ackpayload enable
    while (1) {
        for (i = 0; i < 64; i++) { // clear buffer
            _rxdata[i] = 0;
        }
        for (i = 0; i < 32; i++) { //ack payload example data
            _txackdata[i] = 32-i;
        }
        _txackdata[0] = 31;

        HW2000B_write_reg(0x21, 0x0080); //RX enable，若需写入 FIFO 数据需先打开接收

        //提前写入 ack payload  然后再通过使能接收 FIFO 开始接收
        HW2000B_write_reg(0x36, 0x0000); //FIFO0 bypass
        HW2000B_write_reg(0x37, 0x0000); //FIFO1 bypass
        HW2000B_write_reg(0x3B, 0x4000); //ACK FIFO clear
        HW2000B_write_fifo(0x34, _txackdata, _txackdata[0]+1); // write ack data
        HW2000B_write_reg(0x38, 0x0180); //ACKFIFO0 config
        HW2000B_write_fifo(0x35, _txackdata, _txackdata[0]+1); // write ack data
        HW2000B_write_reg(0x39, 0x0180); //ACKFIFO1 config
        //需往 ACKFIFO0 和 ACKFIFO1 写入同样数据，由状态机自动选取发送数据

        HW2000B_write_reg(0x36, 0x0080); //FIFO0 enable

        delay_ms(5); //延时 5ms
```

```
        _reg = HW2000B_read_reg(0x3D);  //Waiting for rx end，也可用 IRQ 中断方式实现
        while (!(_reg & 0x0001)) {
            delay_ms(5); //延时 5ms
            _reg = HW2000B_read_reg(0x3D);
        }

        _reg = HW2000B_read_reg(0x36);
        if (!(_reg & 0x2000)) {  //CRC 校验
            HW2000B_read_fifo(0x32, _data, 1);
            HW2000B_read_fifo(0x32, &_data[1], _data[0]);
        }

        HW2000B_write_reg(0x38, 0x0080); //clear ackfifo0_ocupy
        HW2000B_write_reg(0x39, 0x0080); //clear ackfifo1_ocupy
        HW2000B_write_reg(0x3D, 0x8808); //Clear INT0, ACKINT0，ACKINT1
        HW2000B_write_reg(0x21, 0x0000); //RX disable
    }
```

5.13 HW2000B POWER DOWN

以下配置可使 HW2000B 进入 POWER DOWN 状态。

```
    void hw2000b_power_down(void)
    {
        uint16_t reg_value;

        HW2000B_write_reg(0x21, 0x0000); //TX RX disable
        reg_value = HW2000B_read_reg(0x23);
        HW2000B_write_reg(0x23, 0x8080 | reg_value); //power down enable,0x23 寄存器
                                                     //第【15】位，置 1，第【7】位，置 1
    }
```

5.14 HW2000B POWER ON

以下配置可实现 HW2000B 从 POWER DOWN 状态唤醒。

```
    void hw2000b_power_on(void)
    {
        uint16_t reg_value;

        reg_value = HW2000B_read_reg(0x23);
        HW2000B_write_reg(0x23, 0x7FFF & reg_value); //从 power down 唤醒，0x23 寄存
```

```
                                                    //器第【15】位，置 0
}
```

5.15 HW2000B 软复位

以下配置可实现 HW2000B 的软复位操作。需要注意的是，软复位只能复位 HW2000B 的内部状态机，并不复位寄存器。软复位后，芯片无需重新初始化，但 0x4C 写保护寄存器会复位，导致写保护打开，被写保护的寄存器（数据手册上未介绍的内部寄存器，如 0x01 寄存器等）不能被读写。若用户有配置内部寄存器的需求，软复位后，0x4C 寄存器需要配置为 0x55AA。

```
void hw2000b_sft_rst(void)
{
    uint16_t reg_value;

    reg_value = HW2000B_read_reg(0x23);
    HW2000B_write_reg(0x23, 0x4000 | reg_value); //软复位使能，0x23 寄存器第【14】
                                                 //位，置 1
    delay_ms(1);
    HW2000B_write_reg(0x23, 0xBFFF & reg_value);//从软复位状态恢复，0x23 寄存器第
                                                 //【14】位，置 0
    delay_ms(1);

    //HW2000B_write_reg(0x4C, 0x55AA); //根据用户需求配置。软复位导致写保护寄存器
                                       //被复位，即写保护被打开使能，重新关闭写保护寄存器
}
```

5.16 HW2000B 发射功率配置例程

以下例程可实现 HW2000B 的发射功率配置。

```
void hw2000b_set_tx_power_8(void)     //8dBm
{
    HW2000B_write_reg(0x4C, 0x55AA);
    HW2000B_write_reg(0x0B, 0x0873);
}

void hw2000b_set_tx_power_7(void)     //7dBm
{
    HW2000B_write_reg(0x4C, 0x55AA);
    HW2000B_write_reg(0x0B, 0x086B);
}
```

```
void hw2000b_set_tx_power_6(void)      //6dBm
{
      HW2000B_write_reg(0x4C, 0x55AA);
      HW2000B_write_reg(0x0B, 0x0862);
}

void hw2000b_set_tx_power_5(void)      //5dBm
{
      HW2000B_write_reg(0x4C, 0x55AA);
      HW2000B_write_reg(0x0B, 0x085F);
}

void hw2000b_set_tx_power_4(void)      //4dBm
{
      HW2000B_write_reg(0x4C, 0x55AA);
      HW2000B_write_reg(0x0B, 0x085B);
}

void hw2000b_set_tx_power_2(void)      //2dBm
{
      HW2000B_write_reg(0x4C, 0x55AA);
      HW2000B_write_reg(0x0B, 0x0855);
}

void hw2000b_set_tx_power_0(void)      //0dBm
{
      HW2000B_write_reg(0x4C, 0x55AA);
      HW2000B_write_reg(0x0B, 0x0851);
}

void hw2000b_set_tx_power_negative_5(void)     //-5dBm
{
      HW2000B_write_reg(0x4C, 0x55AA);
      HW2000B_write_reg(0x0B, 0x084A);
}

void hw2000b_set_tx_power_negative_10(void)   //-10dBm
{
```

```
    HW2000B_write_reg(0x4C, 0x55AA);
    HW2000B_write_reg(0x0B, 0x0845);
}

void hw2000b_set_tx_power_negative_15(void)  //-15dBm
{
    HW2000B_write_reg(0x4C, 0x55AA);
    HW2000B_write_reg(0x0B, 0x0843);
}

void hw2000b_set_tx_power_negative_20(void)  //-20dBm
{
    HW2000B_write_reg(0x4C, 0x55AA);
    HW2000B_write_reg(0x0B, 0x0842);
}

void hw2000b_set_tx_power_negative_25(void)  //-25dBm
{
    HW2000B_write_reg(0x4C, 0x55AA);
    HW2000B_write_reg(0x0B, 0x0841);
}

void hw2000b_set_tx_power_negative_40(void)  //-40dBm
{
    HW2000B_write_reg(0x4C, 0x55AA);
    HW2000B_write_reg(0x0B, 0x0840);
}
```

第 6 章　芯片测试

HW2000B 的接收灵敏度测试可以使用 BER 或 PER 两种测试方法。BER 即误比特率测试，是常用的接收灵敏度测试方法，通常使用一台带 BER 测试功能的矢量信号发生器，按照一定的操作流程来实现。PER 即误包率测试，通过误包率同样可以测试芯片的接收灵敏度，同时测试了芯片的收包能力。本章节主要介绍 PER 的测试方法。

6.1　PER灵敏度测试

◆ 测试平台

- 测试仪器：Agilent N5182A 矢量信号发生器
- 测试触发板：可产生触发脉冲的 PCB 板（如无线开发底板）
- 测试对象：无线开发底板+ HW2000B 模块（带 SMA 接口）

◆ 测试方法

- 采用丢包率测试法，信号发生器发射 1000 个数据包，计算 HW2000B 模块收包数。
- HW2000B 能接收到 900 个以上数据包的条件下，信号发生器的最低发射功率就是 HW2000B 的接收灵敏度。

◆ 测试步骤

- 用射频同轴线，把 HW2000B 模块的 SMA 接口与信号发生器的 RF 输出口相连。
- 操作无线开发底板，使 HW2000B 进入接收状态。
- 打开信号发生器，选择相应的 HW2000B 数据波形文件，配置好仪器参数，进入触发等待状态。
- 打开触发板，触发信号发生器发射数据包。触发板共产生 1000 个脉冲，激励信号发生器发射 1000 个数据包。
- 记录 HW2000B 的收包数，当收包数达到 900 以上，把信号发生器的最低发射功率作为 HW2000B 的接收灵敏度。

◆ 信号发生器产生的数据包格式

信号发生器产生帧格式的数据包，内容如下：

- 帧格式表

4 Bytes	6 Bytes	4 Bits	3 Bits	21 Bytes	2 Bytes
Preamble	PIPE Address	Trailer	Packet Control	PAYLOAD	CRC

- Preamble（前导码），4 个字节：0x55 0x55 0x55 0x55
- PIPE Addres（同步字），6 个字节：0xE7 0xE7 0xE7 0xE7 0xE7 0xE7
- Trailer，4 个比特：B1010

- Packet Control ，3 个比特：B000（包括 PID，2 比特：B00；NOACK，1 比特：B0）
- PAYLOAD，21 个字节：

  0x14 0xDF 0xC4 0x59 0xBA 0xA5 0x06 0xC5 0x3B 0xD4

  0x32 0x92 0x5C 0xD2 0xA2 0xE4 0xF6 0x41 0x74 0x8A

  0xD2
- CRC，2 个字节：0xF1 、0x9E

第 7 章　芯片故障分析

若两颗芯片不能进行正常收发通讯，首先应该将这两颗芯片分别与已测试过的正常芯片进行收发通讯，确定是发射芯片还是接收芯片的问题。在此前提下，可参照以下内容分析芯片发送或接收故障的原因。

✧ **检查 SPI 读写是否正常**

对寄存器进行读写操作，检查 SPI 驱动程序是否正确。若寄存器写入值与读取值不一致，可使用示波器抓取 SPI 的读写波形，检查 SPI 四根线的电平是否正确，波形时序是否和产品手册的参考时序一致；检查 SPI 通讯速率是否小于给定的最大通讯速率，建议 SPI 速率不要超过 4Mbps。

✧ **软件初始化和收发流程检查**

检查软件对 HW2000B 芯片的寄存器初始化配置是否与参考代码有差异，软件收发流程是否与操作例程有差异。建议先使用最简单的 NOACK 模式进行调试。

✧ **检查 ISM 频段的辐射干扰**

测试 ISM 频段(2400MHz~2483MHz)有无辐射干扰的最直接方法是将 2.4G 天线连接到频谱仪，直接测量该频段内的空中辐射信号。推荐的频谱仪设置参数为 Span＝2300MHz～2500MHz，Ref Amplitude<-50dBm。若空中有较大的干扰信号，建议避开此频点进行通讯测试。

✧ **连续发送模式**

向 FIFO0 内填写 0x55 或 0xAA(64 bytes)，将 0x29 设置为 0x0000，使能发送。观察发射频点是否锁定，与设置值是否一致。

✧ **检查发送频偏**

在连续发送模式下，检查信号的中心频点频偏值是否小于 100KHz（测量值与理论值频率之间的偏差）。

✧ **检查接收频偏**

检查接收频偏值是否小于 100KHz，频偏补偿方法详见 HW2000B 芯片数据手册的“自动频偏校正”章节。

✧ **检查接收本振是否锁定**

判断接收本振是否锁定，需要使用频谱仪测量。频谱仪设置参数为 Span=250KHz，Ref Amplitude=-50dBm。使能芯片接收，若接收频点一直停留在设定值不跳动，则说明已锁定。

✧ **检查晶振是否正常工作**

上电后测试晶振 XTALP 或 XTALN 引脚，若有和晶振频率相同的波形信号，则说明晶振工作正常，否则可能是晶振损坏。

✧ **晶振频偏调整**

调整晶振外接的两个电容，若调整以后的晶振频率值较差，则需更换晶振，选取精度和频率稳定度较好的晶振。

✧ **检查电源电压是否超出范围**

芯片支持的电源电压范围是 2.0V~3.6V，低于 2.1V 会导致通讯不正常，超过 3.6V 可能会导致芯片损坏。

✧ **芯片发送或接收频点为什么与设置值不一致？**

频点设置需在芯片发送或接收状态有效之前完成，否则芯片内部 PLL 可能无法正确锁定。

✧ **写指针在何时需要软件清‘0’ ？**

芯片内状态机写 FIFO（只出现在 PRX 端）指针和 SPI 写 FIFO（只出现在 PTX 端）指针复用同一个指针。当 PTX 与 PRX 在正常收发不切换时，硬件在合适情况下自动清写指针，无需软件参与。但在收发角色切换（PRX 切换为 PTX）写指针主控权发生变化时，需要软件参与在 SPI 写 FIFO 前将写指针清‘0’。

✧ **为什么 PTX 发送完成后中断无法清‘0’ ？**

发送中断置起后，需先将 PTX_FIFOn_OCPY 控制信号清‘0’后再清中断，若只清中断 INTn，FIFOn 仍处于有效状态，会自动进入发送流程，发送完成后中断再次置起。

✧ **为什么同步字长度设为 16bits 很容易误相关？**

同步字长度若设为 16bits，由于长度较短，出现误相关的概率较大。推荐的同步字错误阈值 SYNC_THRES（0x28）设置为 1 或 0。

✧ **收发 ACK 使能时为什么 PTX 发送中断正常置起而 PRX 接收无正常中断？**

ACK 使能时，PTX 发送方中断置起分两种情况：

(1) PTX 正常收到 ACK 信号，通讯成功。

(2) PTX 重传超时，通讯不成功。

PTX 中断置起后，可通过 PTX_FIFOn_FAIL 标志位来区分这两种情况。

ACK 使能时，PRX 在接收 PID 与 CRC 较上一次相同时，将自动弃包而不置起中断。若 PTX 出现重传超时，则 PTX 在下一次发送帧时不累加 PID。

✧ **ACK PAYLOAD 功能使能时，为什么 PTX 的 ACK_INTn 中断标志正常置起，但读取 ACKFIFO 值不正确或仍保留上一次收到的值？**

PTX 在收到零长度的 ACK PAYLOAD（即只有 ACK 信号）时，不对 ACKFIFO 进行写操作，但标志位 ACKINTn_W_ACKPAY 将会置起。上层软件在读 0x3D 寄存器时，可依据此标志位确认此次收发流程有无 ACK PAYLOAD。

✧ **为什么收发双方 PIPE Address 配置一致时，会出现 PRX 的 PIPE 指示位（PRX_FIFOn_PIPE）所指示的 PIPE 与发送端的 PIPE 不一致的现象？**

各 PIPE Address（0x40~0x47）设置值之间的码间距需大于接收同步字允许的错误个数阈值（SYNC_THRES），否则，接收各 PIPE 容易出现误同步现象。

✧ **为什么 PRX 接收 CRC 正确，但接收端 FIFO 的读取值与 PTX 发送端 FIFO 写入值不一致？**

(1) PTX 在写 FIFO 时，SPI 可能受到干扰而误写，则芯片内部硬件会根据误写的 FIFO 值自动生成 CRC，PRX 收到误写的值则会出现该现象。

(2) PRX 在读 FIFO 时，SPI 可能受到干扰而误读。

✧ **ACK PAYLOAD 功能使能时，为什么 PRX 的 INTn 中断标志正常但无 ACK_INTn 中断标志？**

(1) PRX 在返回 ACK PAYLOAD 之后，只有再收到新帧（PID 变化的帧）时，确认 PTX 成功收取到上一帧的 ACK PAYLOAD，才会置起中断标志位 ACK_INTn。

(2) PRX 可能无匹配的 ACKFIFO，故只向 PTX 返回了 ACK。

（详见 HW2000B 芯片数据手册的“ACK 带 ACK PAYLOAD”章节）。

✧ **CE、PD_CTRL、SFT_RST 区别**

(1) 正常工作时，需将 CE 引脚拉高，使能 HW2000B 芯片。若拉低 CE 引脚，将全局复位 HW2000B 芯片，包括复位芯片内部各状态信号与寄存器。

(2) PD_CTR(0x23[15])为芯片进入 POWER DOWN 模式使能信号，仅控制芯片进入掉电模式，寄存器状态保持并可读写。

(3) SFT_RST(0x23[14])为软件复位使能信号，仅复位芯片内部状态信号，并不复位内部寄存器，即 INT 中断信号等状态变量会被复位。此外，SFT_RST 使能后会将写保护关闭（0x4C 寄存器的值变为 0xFFFF），导致功率配置寄存器（0x0B）等未开放寄存器不能被修改。

✧ **接收/发射模式下，无法更换频点**

若要动态更换频点，则需先关闭接收或发射功能，再进行频点更换。

✧ **发射使能后填写 FIFO，写入值出现错误**

(1) 发射端 TX 使能(DBUS_TX=’1’)后需延时 5us，等待芯片从 SLEEP 状态切换为 IDLE 状态后，才能进行写 FIFO 操作，否则写入 FIFO 的数据可能出错。

(2) 在 SPI 速率≤1Mbps 的情况下，无需上述延时操作，否则，必须添加延时操作。

(3) 在 TX 使能操作后，需将 FIFO 指针清零后再进行写 FIFO 操作，相当于在 TX 使能后加入了延时时间，故无需再添加延时操作。

✧ **内部寄存器 0x06 读写不一致问题**

内部寄存器 0x06 为只读寄存器，其低 8 位是只读位，故其值会动态变化。

✧ **接收端清中断操作**

在连续接收（接收使能一直打开）的情况下，接收端在接收完数据后，需判断 CRC 校验是否成功。但无论校验是否成功，都必须清接收中断，否则接收状态将暂停。

✧ **ACK 使能时，不支持 FEC 功能**

FEC 功能只在硬件链路控制和 ACK 不使能的情况下有效。

✧ **ACK PAYLOAD 功能使能时的 PREAMBLE 长度最小值为 6 bytes**

在 ACK PAYLOAD 功能使能时，PREAMBLE 长度的最小设置值应为 6 bytes，否则 PRX 会出现丢包或错包现象，PTX 会出现重发超时或 ACK 不带 PAYLOAD 的现象。

✧ **收发未使能时 IRQ 中断标志位无法清‘0’**

在完成收发流程中断置起后，若关闭发送/接收使能，芯片将进入 SLEEP 模式，故无法进行 IRQ 中断标志位的清‘0’操作。所以，可在重新使能发送/接收后清‘0’，或使用 SFT_RST 软复位功能来清‘0’。