文档编号：AN1045

上海东软载波微电子有限公司

应用笔记

HW3000 应用注意事项

修订历史

版本	修订日期	修改概要
V1.0	2019-10-18	初版公开发布
V1.1	2020-03-09	增加 5.7 节直接频点模式跳频接收例程 增加第 6 章芯片测试章节 版本号从 V1.0 升级到 V1.1
V1.2	2020-07-14	增加状态切换说明章节（4.11） 增加低功耗说明章节（4.12） 增加 IO 模拟 SPI 驱动章节（5.4） 修改数据接收例程（5.6） 版本号从 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 HW3000 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《AN1013_HW3000 User Guide》和《AN1015_HW3000 Programming Guide》两份应用笔记，补充注意事项和常见问题，更改和增加的内容包括：

(1) 工作模式（2.1 小节）
(2) 直接收发 DIRECT 模式（2.4 小节）
(3) BANK0 与 BANK1 的说明（3.1 小节）
(4) 频点设置解析和频点直接设置模式注意事项（3.3 小节）
(5) MISO 引脚无高阻态（4.1 小节）
(6) 空中数据流方向（4.5 小节）
(7) 直接 FIFO 模式下 RSSI 如何读取（4.7 小节）
(8) 收发频偏问题处理（4.6 小节）
(9) PDN 状态下 IRQ 引脚处理（4.9 小节）
(10) 状态切换说明（4.11 小节）
(11) 低功耗说明（4.12 小节）
(12) IO 模拟 SPI 驱动（5.4 小节）
(13) 收据接收例程（5.6 小节）
(14) 跳频接收（5.7 小节）
(15) 灵敏度测试（6.1 小节）

第 2 章　工作模式

2.1　工作模式概述

HW3000 芯片主要有 POWER DOWN、DEEP SLEEP、SLEEP、IDLE、TX、RX 六个工作模式，各个工作模式之间的切换，详见 HW3000 数据手册的“芯片状态控制”章节。

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，并且严格按照芯片数据手册中“HW3000 状态控制示意图”中的状态来进行切换。

例如，把 HW3000 从 TX 模式切换到 RX 模式，必须先关闭 TX 模式，使芯片进入 IDLE 模式后，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　增强型和直接 FIFO 帧结构

HW3000 芯片支持增强型帧结构和直接 FIFO 帧结构，可通过 PACK_LENGTH_EN 寄存器配置：

- 增强型帧结构可兼容国家电网电力用户用电信息采集系统通信协议帧结构，并支持自动 ACK 功能；
- 直接 FIFO 帧结构配置灵活，可兼容增强型帧结构、802.15.4g 协议等帧结构。

2.2.1　增强型帧结构

增强型帧结构模式需设置 PACK_LENGTH_EN 为‘1’，该模式的帧结构如图 2-1 所示。

字节数：4~1023	2~6	1	1	1	1	0~252	2
前导码	帧分隔符	帧长	信道索引	标准识别号	帧头校验码	物理层载荷	帧校验序列
SHR		PHR				PSDU	FCS

图 2-1　增强型帧结构示意图

◆ 帧长配置

发送端可通过 AUTO_LEN_CALC(0x09)寄存器配置，发送帧长信息由硬件自动生成或由软件填写。

若设置 AUTO_LEN_CALC 为‘1’，硬件自动依据发送 FIFO 内所填写的 PSDU 字节数加上 3（即加上信道索引、标准识别号、帧头校验码所占的 3 个字节）作为发送帧的帧长信息。

若设置 AUTO_LEN_CALC 为‘0’，在发送之前需由软件将 PSDU 的长度信息填写至 TXPSDU_LEN(0x09)寄存器，硬件依据 TXPSDU_LEN 寄存器的值加上 3（即加上信道索引、标准识别号、帧头校验码所占的 3 个字节）作为发送帧的帧长信息。

接收端将成功接收的数据帧帧长信息存入 RX_PHR_PSDU_LEN(0x0A)寄存器。

◆ 信道索引

发送端需在发送之前，将信道索引写入 TX_PHR_CHANNEL_INDEX(0x08)寄存器。

接收端将成功接收的数据帧信道索引存入 RX_PHR_CHANNEL_INDEX(0x0A)寄存器。

◆ 标准识别号

发送端需在发送之前，将标准识别号写入 TX_PHR_STD_IDEF(0x08)寄存器。

接收端将成功接收的数据帧标准识别号存入 RX_PHR_STD_IDEF(0x0B)寄存器。

◆ 帧头校验

发送端可通过 AUTO_PHR_VERIFY(0x07)寄存器配置帧头校验信息，由硬件自动生成或由软件生成。

若设置 AUTO_PHR_VERIFY 为‘1’，硬件依据发送帧长、信道索引、标准识别号 3 个字节自动生成的校验码作为发送帧的帧头校验信息。

若设置 AUTO_PHR_VERIFY 为‘0’，在发送之前需由软件填写 TX_PHR_VERIFY(0x07)寄存器，硬件依据 TX_PHR_VERIFY 寄存器的值作为发送帧的帧头校验。

接收端将成功接收的数据帧帧头校验信息存入 RX_PHR_VERIFY(0x0B)寄存器。

◆ 物理层载荷（PSDU）

PSDU 部分存放在 FIFO 中，增强型帧结构模式下最大支持的数据长度为 252 字节。

◆ 帧校验序列（FCS）

发送端可通过 LEN1_CRCSEL(0x02)寄存器配置帧校验序列，由硬件自动生成或由软件生成。

若设置 LEN1_CRCSEL 为‘0’，硬件自动生成帧校验序列。

若设置 LEN1_CRCSEL 为‘1’，在发送之前需由软件填写 CRCVAL(0x12)寄存器，硬件依据 CRCVAL 寄存器的值作为发送帧的帧校验序列。

接收端将成功接收的数据帧帧校验序列存入 CRCVAL 寄存器。

注：ACK 功能使能时，不支持帧头校验与帧校验序列软件生成模式。

增强型帧结构的收发流程如图 2-2 所示。

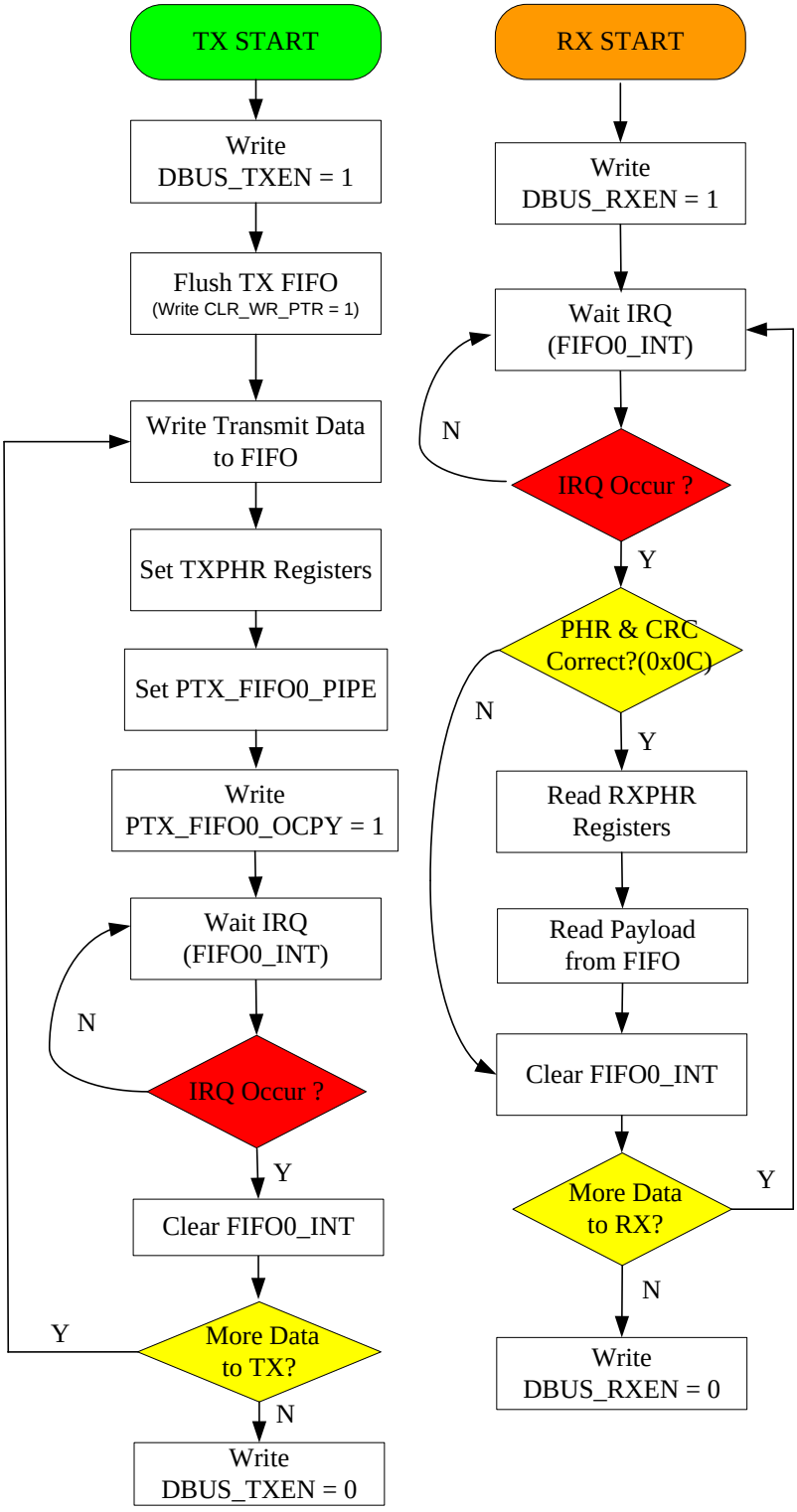

图 2-2 增强型帧结构下的收发流程示意图

HW3000 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 2-3 所示）。用户可在再次使能发送之前插入延时，控制重发时间间隔或更改发送频点。

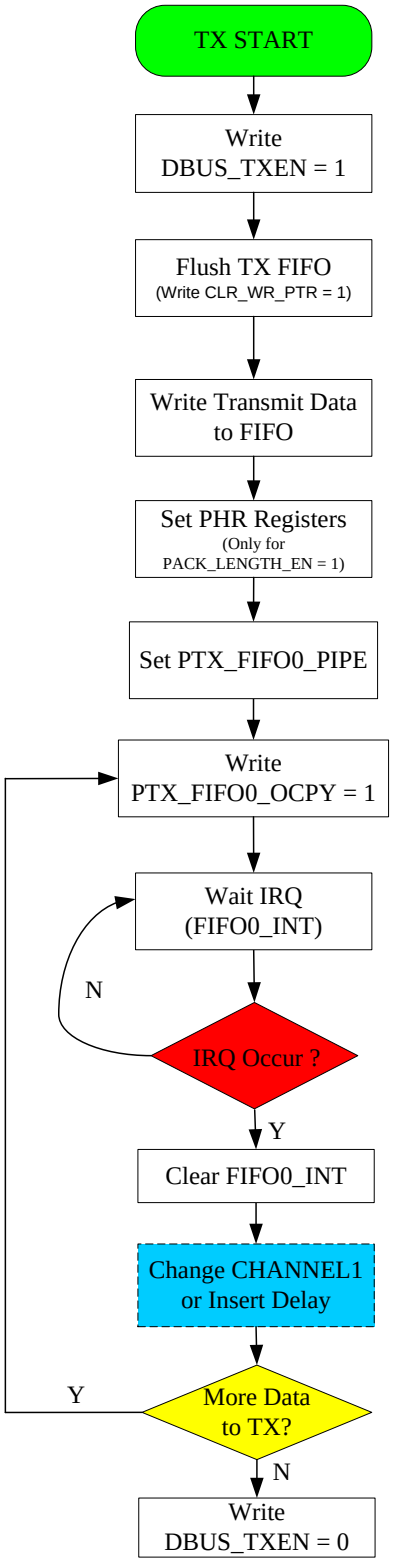

图 2-3 FIFO 数据包重复发送控制流程

注：频点设置需在芯片发送或接收状态有效之前完成。

2.2.2 直接 FIFO 帧结构模式

直接 FIFO 帧结构模式需设置 PACK_LENGTH_EN 为‘0’，该模式的帧结构如图 2-4 所示。

此模式不支持 ACK、硬件 CRC 校验与 FEC 功能。除前导码 PREAMBLE 和帧分隔符 SFD 部分，都需用软件方式写入 FIFO，写入内容与顺序可依据收发双方的约定灵活设置。

在直接 FIFO 模式下，发送和接收各有两种模式来配置。通过 0x02 寄存器的 LEN0_TXMODE、LEN0_RXMODE 控制位来具体实现，如表 2-1 和表 2-2 所示：

TX 模式	LEN0_TXMODE	描述
TX_MODE0	0	发送长度按 LEN0_PKLEN 设置值发送
TX_MODE1	1	读写指针相等时发送结束

表 2-1 直接 FIFO 下 TX 模式

RX 模式	LEN0_RXMODE	描述
RX_MODE0	0	芯片自动识别包长度，要求发射端需按规定的帧长信息位置、长度填充发送 FIFO
RX_MODE1	1	接收长度按 LEN0_PKLEN 设置值接收

表 2-2 直接 FIFO 下 RX 模式

直接 FIFO 的帧结构如下图所示：

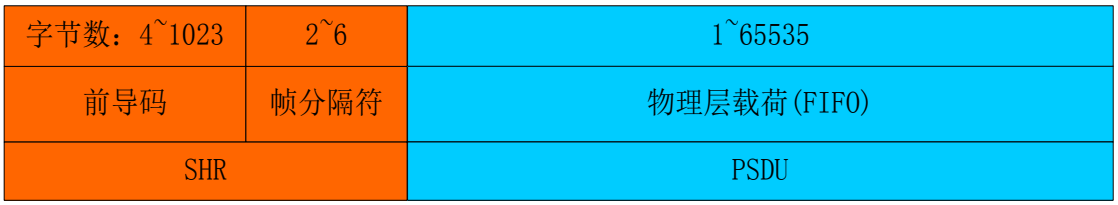

图 2-4 直接 FIFO 模式帧结构示意图

直接 FIFO 帧结构模式下，芯片提供接收长度自动识别功能，具体设置方法可参考《HW3000_Datasheet_C》8.2 章节的描述。

若收发长度小于等于 256 字节，且需设置 LEN0_TXMODE 及 LEN0_RXMODE 为芯片自动识别包长，其收发流程如图 2-5 所示。

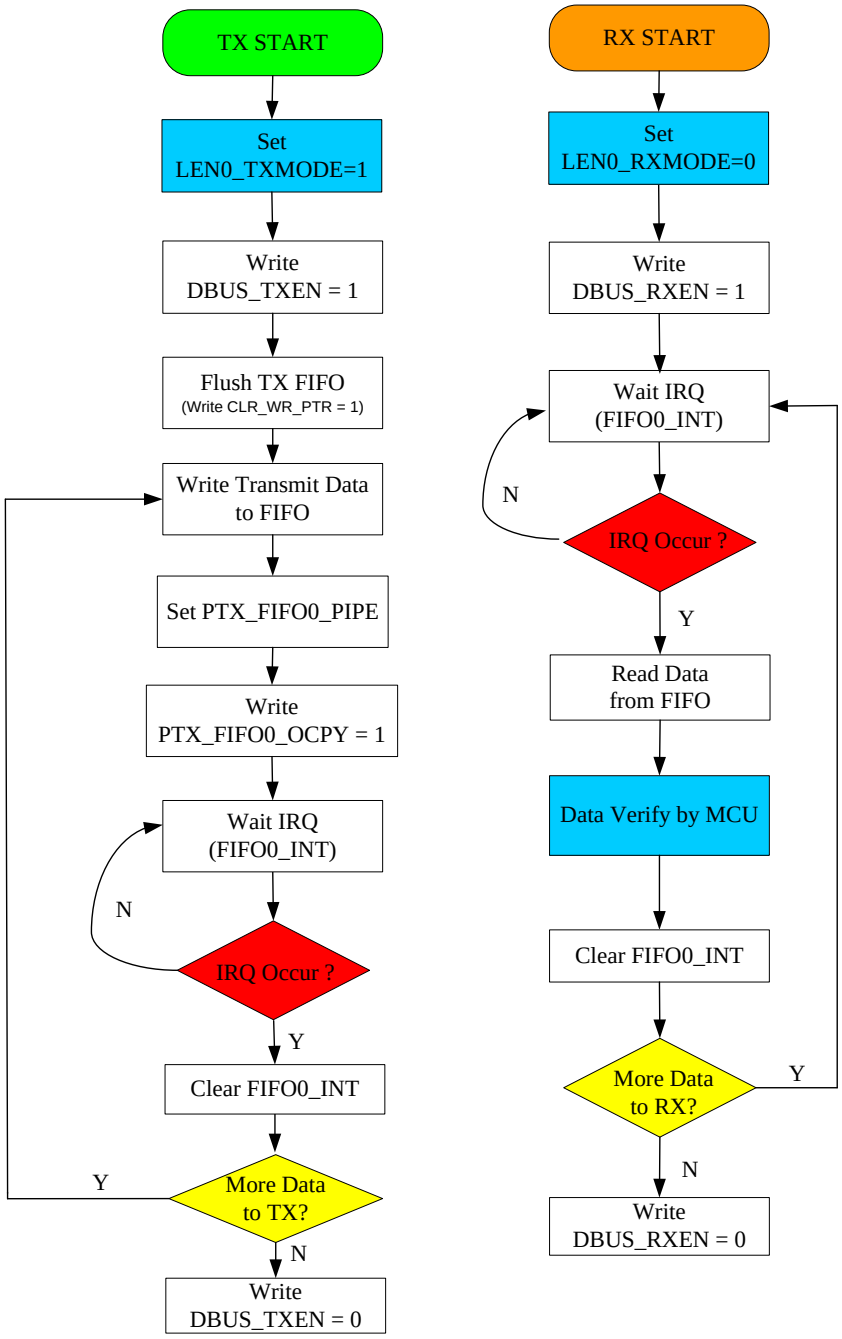

图 2-5　直接 FIFO 帧结构下自动识别包长的收发流程示意图

若收发长度小于等于 256 字节，且 LEN0_RXMODE 设置为软件设置接收长度，假设收发约定 FIFO 内第一个字节代表包长度信息，其接收流程如图 2-6 所示。

接收使能之前，先设定一个较小的半满阈值，确保半满中断置起时，可从 FIFO 内指定位置读取接收长度。软件获取接收长度后，需配置 LEN0_PKLEN 寄存器，硬件依据 LEN0_PKLEN 寄存器的设定值自动完成接收过程。

使能之前，需要配置 LEN0_PKLEN 为一个较大值，防止软件处理过慢时出现 FIFO0_INT 误中断。

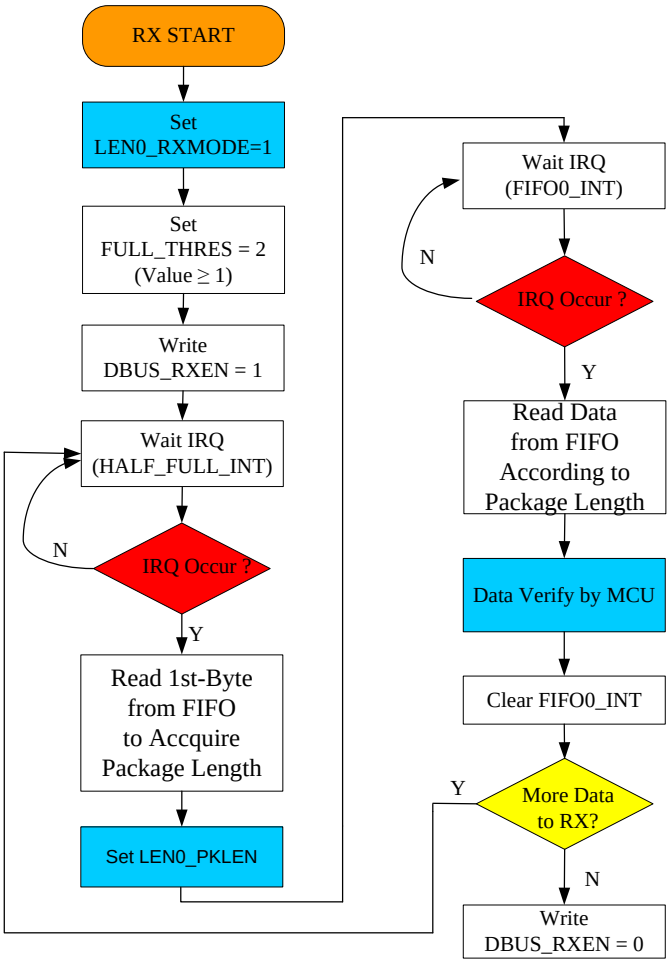

图 2-6　直接 FIFO 帧结构模式下软件设置包长的接收流程示意图

若收发长度大于 256 字节时，软件需依据 FIFO 的半空或半满中断标志，完成对 FIFO 的写入与读取操作，配合物理层硬件完成对 FIFO 内数据的发送与接收。

假设收发约定 FIFO 内第一个 byte 代表包长度信息，LEN0_RXMODE 设置为接收自动识别包长信息，其收发流程如图 2-7 所示。

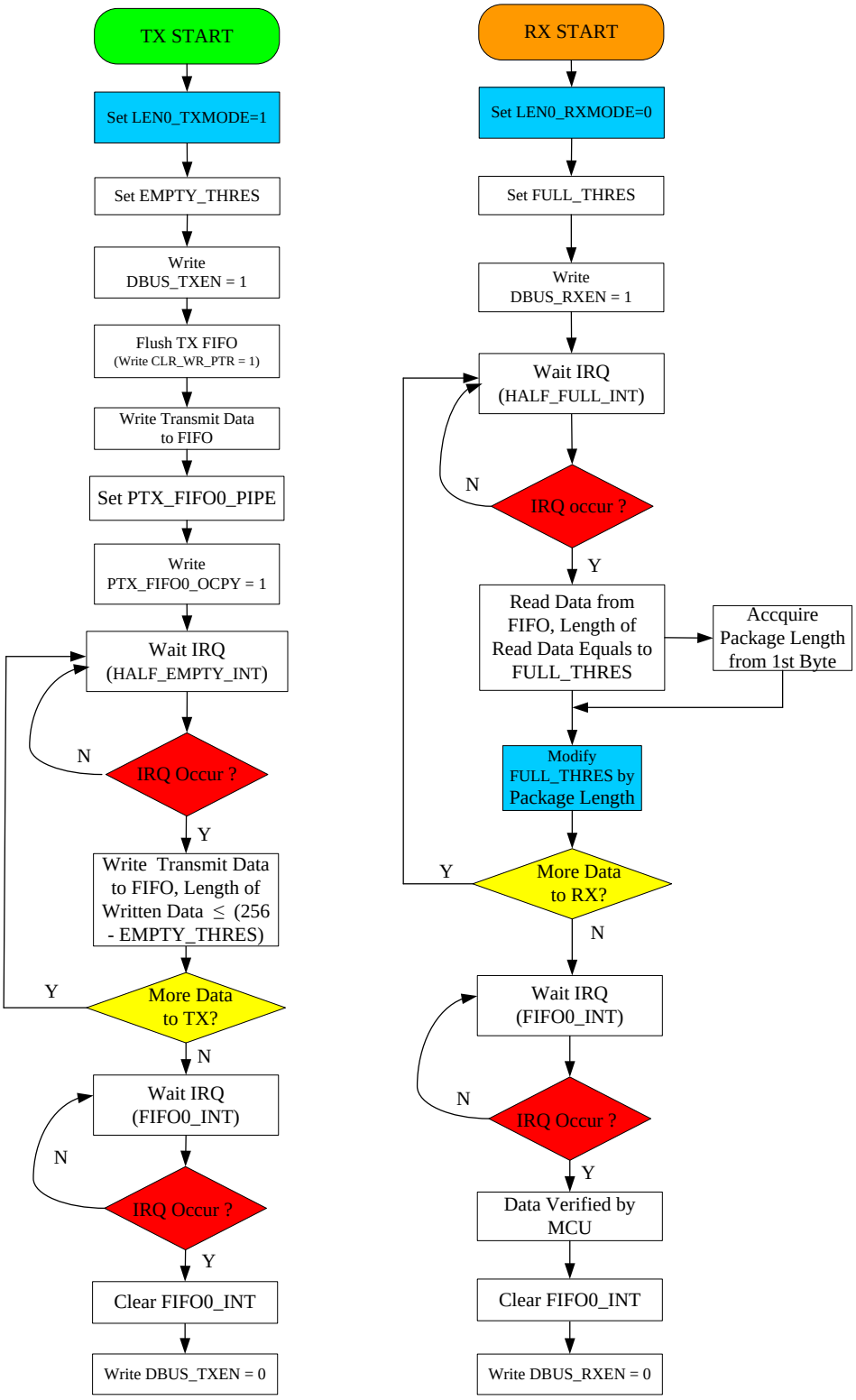

图 2-7　直接 FIFO 帧结构收发(>256 字节)流程示意图

PRX 在接收使能之前可设定一较小的半满阈值 FULL_THRES，保证响应 HALF_FULL_INT 中断时软件可从 FIFO 内即时获取接收包长信息，根据接收包长信息再由软件调整半满阈值 FULL_THRES 以简化接收操作流程。

注：可通过 EMPTY_THRES，FULL_THRES 寄存器设置半空半满阈值，设置值需考虑 SPI 接口的访问速度与芯片配置的数据速率。

2.3 ACK 功能

ACK 功能的应用仅针对增强型帧结构模式，功能介绍详见《HW3000_Datasheet_C》第 5 章说明，ACK 使能情况下收发流程如图 2-8 所示。

若使能 PIPE0 ACK 功能（P0_ACKEN = '1'），PTX 在接收 ACK 等待时间 AUTO_RXACK_TIME（0x45）内没有成功接收到 PRX 发送的 ACK 帧，PTX 将自动重传上一帧数据包。PTX 读取中断 FIFO0_INT 之后，需检查 FIFO0_MAX_RETX（0x0F）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 使能情况下，接收 PHR 校验出错或 CRC 出错将进入自动重收流程，FIFO0_INT 中断标志位置起后，软件无需再检测 PRX_PHR_ERR 与 PRX_CRC_ERR 标志位。

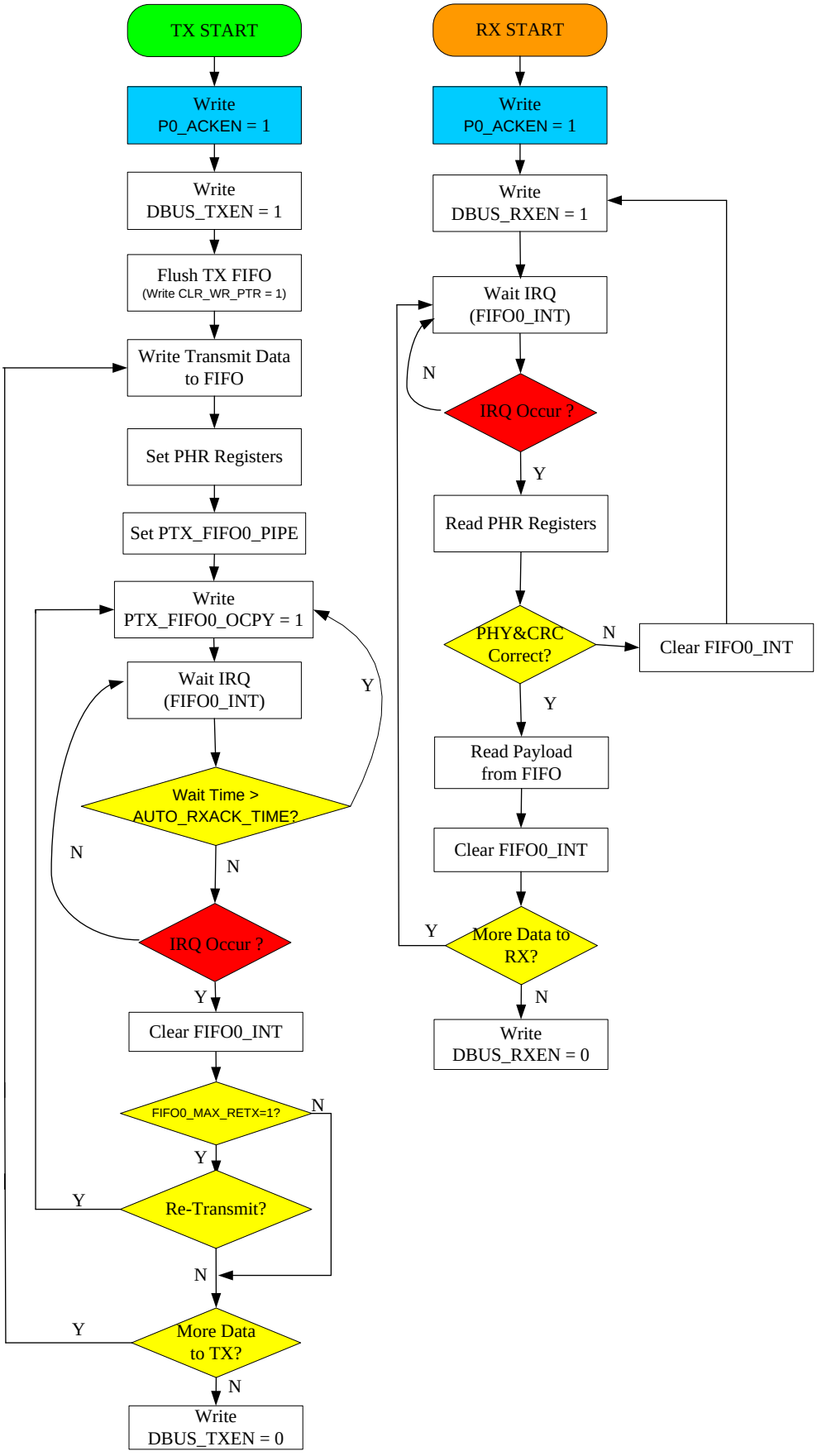

图 2-8 ACK 功能使能收发控制流程

2.4 直接收发 DIRECT 模式

芯片提供直接收发 DIRECT 模式，可用于灵敏度 BER 测试、数据透传等应用，具体配置方法如下。

2.4.1 Direct RX 模式

Direct RX 配置如表 2-3 所示：

寄存器地址（Hex）	寄存器配置（Hex）	说明
1D	8584	GPIO0 配置为 RX DATA OUTPUT GPIO1 配置为 RX CLOCK OUTPUT
1C	110F	使能 Direct 模式
01	0080	RX 接收使能
延时 300us 以上		
1C	130F	Direct 控制信号使能

表 2-3 Direct RX 模式配置

2.4.2 Direct TX 模式

Direct TX 配置如表 2-4 所示：

寄存器地址（Hex）	寄存器配置（Hex）	说明
1D	E382	GPIO0 配置为 TX DATA INPUT GPIO1 配置为 TX CLOCK OUTPUT
1C	110F	使能 Direct 模式
01	0100	TX 发送使能
延时 300us 以上		
1C	130F	Direct 控制信号使能
等待 PAUP_DONE(0x1B)指示位置为’1’后，从 GPIO 输入发送数据		

表 2-4 Direct TX 模式配置

2.5 连续发送模式

在直接收发模式的情况下，芯片支持内部数据的连续发送方式。

2.5.1 0101 序列连续发送

芯片使能 REP_MODE（0x1C），支持连续发送“0101”数据序列，具体配置如表 2-5 所示：

寄存器地址（Hex）	寄存器配置（Hex）	说明
1C	190F	使能 Direct 模式及发送“0101”序列
01	0100	TX 发送使能
延时 300us 以上		
1C	1B0F	Direct 控制信号使能

表 2-5 连续发送“0101”序列配置

2.5.2 PN9 序列连续发送

芯片使能 PN9_MODE（0x1C），支持连续发送 PN9 数据序列，具体配置如表 2-6 所示：

寄存器地址（Hex）	寄存器配置（Hex）	说明
1C	150F	使能 Direct 模式及发送 PN9 序列
01	0100	TX 发送使能
延时 300us 以上		
1C	170F	Direct 控制信号使能

表 2-6 连续发送 PN9 序列配置

2.6 单载波发送模式

芯片提供单载波（CW）发送模式，便于频点测试与发送功率测试，具体配置如表 2-7 所示：

寄存器地址（Hex）	寄存器配置（Hex）	说明
02	E000	使能单载波模式
2C	0000	Deviation 配置为 0 Hz
01	0100	TX 发送使能
70	5555	FIFO 数据填充
0C	0001	指示发送 FIFO 有效

表 2-7 单载波发送配置

第 3 章　寄存器配置

HW3000 支持的数据速率范围是 1.2Kbps~100Kbps，支持 20MHz 和 26MHz 两种晶振，支持 5 个典型频段：315/433/779/868/915MHz，频段范围内各个频点可配。

3.1　BANK0 和 BANK1

HW3000 寄存器分为 BANK0 和 BANK1 两个寄存器列表，二者有多个寄存器地址重合，需要注意的是，地址重合的寄存器分属不同 BANK，是完全不同的两个寄存器，用户在使用时一定不能混淆：

- BANK0 寄存器列表对外开放，用户可在数据手册寄存器章节中查找各个寄存器的功能介绍。
- BANK1 寄存器列表不对外开放，包含的是内部校准寄存器，为防止改错，用户必须按照 SDK 例程包中的配置来处理。

两个寄存器列表通过 0x4C 寄存器进行切换，只有在 0x4C 寄存器配置为 0x55AA 时，BANK1 才能进行读写操作，配置为其他值时，均代表关闭 BANK1，打开 BANK0。

3.2　速率配置

HW3000 支持的数据速率范围是 1.2Kbps~100Kbps，以下为 20MHz 晶振下 433MHz 频段典型速率的寄存器初始化配置。

关于频点，帧格式配置等，用户可根据实际需求配置，也可参考 SDK 例程包。

3.2.1　1.2Kbps 速率配置

Bank 切换	Register Address (Hex)	Default Value (Hex)	Recommended Value (@20MHz Crystal Frequency) (Hex)
Bank1 使能	4C	-	55AA
Bank1（寄存器不对外开放）	03	050A	0508
	11	8630	C630
	14	1915	1935
	40	000F	000F
	41	001F	001F
	42	83FB	83FB
	43	3C24	3C24
	17	F91C	F6C2
	1B	1F99	1371
	1C	1F89	1351
	1D	1F81	0B51
	1E	1A81	2F71
	1F	1581	2F51
	20	3A61	2E51

Bank 切换	Register Address (Hex)	Default Value (Hex)	Recommended Value (@20MHz Crystal Frequency) (Hex)
	21	3961	2E31
	22	2D61	4B31
	23	2C61	4731
	24	4C81	4631
	25	4861	4531
	26	4461	4431
	27	4441	6131
	28	6061	6031
	29	6041	6011
	2A	6021	6009
	51	0003	001B
	55	8002	8003
	56	4555	4155
	62	77ED	70ED
Bank0 使能	4C	-	5555
Bank0（寄存器对外开放）	11	-	8000
	26	2CCD	2E35
	1C	1046	100F
	25	1201	5201
	32	0051	0009
	33	00EC	00D5
	2E	0022	0006
	2C	003F	0007
	35	3332	3312
	40	003F	FF3F

表 3-1　1.2Kbps 速率寄存器初始化设置

3.2.2　10Kbps 速率配置

Bank 切换	Register Address (Hex)	Default Value (Hex)	Recommended Value (@20MHz Crystal Frequency) (Hex)
Bank1 使能	4C	-	55AA
Bank1（寄存器不对外开放）	03	050A	0508
	11	8630	C630
	14	1915	1935
	14	1915	1935
	40	000F	0008
	41	001F	0010

Bank 切换	Register Address (Hex)	Default Value (Hex)	Recommended Value (@20MHz Crystal Frequency) (Hex)
	42	83FB	82D8
	43	3C24	3D38
	17	F91C	F6C2
	1B	1F99	1371
	1C	1F89	1351
	1D	1F81	0B51
	1E	1A81	2F71
	1F	1581	2F51
	20	3A61	2E51
	21	3961	2E31
	22	2D61	4B31
	23	2C61	4731
	24	4C81	4631
	25	4861	4531
	26	4461	4431
	27	4441	6131
	28	6061	6031
	29	6041	6011
	2A	6021	6009
	40	000F	0008
	41	001F	0010
	42	83FB	82D8
	43	3C24	3D38
	51	0003	001B
	55	8002	8003
	56	4555	4155
	62	77ED	70ED
Bank0 使能	4C	-	5555
Bank0（寄存器对外开放）	11	-	8000
	26	2CCD	2E35
	1C	1046	100F
	25	1201	5201
	32	0051	0051
	33	00EC	00EC
	2E	0022	0025
	2C	003F	0052
	35	3332	3312
	40	003F	FF3F

表 3-2　10Kbps 速率寄存器初始化设置

3.2.3 50Kbps 速率配置

Bank 切换	Register Address (Hex)	Default Value (Hex)	Recommended Value (@20MHz Crystal Frequency) (Hex)
Bank1 使能	4C	-	55AA
Bank1（寄存器不对外开放）	03	050A	0508
	11	8630	C630
	14	1915	1915
	40	000F	000F
	41	001F	001F
	42	83FB	83FB
	43	3C24	3C24
	17	F91C	F6C2
	1B	1F99	1371
	1C	1F89	1351
	1D	1F81	0B51
	1E	1A81	2F71
	1F	1581	2F51
	20	3A61	2E51
	21	3961	2E31
	22	2D61	4B31
	23	2C61	4731
	24	4C81	4631
	25	4861	4531
	26	4461	4431
	27	4441	6131
	28	6061	6031
	29	6041	6011
	2A	6021	6009
	51	0003	001B
	55	8002	8003
	56	4555	4155
	62	77ED	70ED
Bank0 使能	4C	-	5555
Bank0（寄存器对外开放）	11	-	8000
	26	2CCD	2E35
	1C	1046	100F
	25	1201	5201
	32	0051	0199
	33	00EC	009A
	2E	0022	0037

Bank 切换	Register Address (Hex)	Default Value (Hex)	Recommended Value (@20MHz Crystal Frequency) (Hex)
	2C	003F	0052
	35	3332	3312
	40	003F	FF3F

表 3-3　50Kbps 速率寄存器初始化设置

3.2.4　100Kbps 速率配置

Bank 切换	Register Address (Hex)	Default Value (Hex)	Recommended Value (@20MHz Crystal Frequency) (Hex)
Bank1 使能	4C	-	55AA
Bank1（寄存器不对外开放）	03	050A	0508
	11	8630	C630
	14	1915	1915
	40	000F	000F
	41	001F	001F
	42	83FB	83FB
	43	3C24	3C24
	17	F91C	F6C2
	1B	1F99	1371
	1C	1F89	1351
	1D	1F81	0B51
	1E	1A81	2F71
	1F	1581	2F51
	20	3A61	2E51
	21	3961	2E31
	22	2D61	4B31
	23	2C61	4731
	24	4C81	4631
	25	4861	4531
	26	4461	4431
	27	4441	6131
	28	6061	6031
	29	6041	6011
	2A	6021	6009
	51	0003	001B
	55	8002	8003
	56	4555	4155
	62	77ED	70ED

Bank 切换	Register Address (Hex)	Default Value (Hex)	Recommended Value (@20MHz Crystal Frequency) (Hex)
Bank0 使能	4C	-	5555
Bank0（寄存器对外开放）	11	-	8000
	26	2CCD	2E35
	1C	1046	100F
	25	1201	5201
	32	0051	0333
	33	00EC	0033
	2E	0022	0078
	2C	003F	00A4
	35	3332	3312
	40	003F	FF3F

表 3-4 100Kbps 速率寄存器初始化设置

3.3 频段配置

HW3000 支持 5 个典型频段：315/433/779/868/915MHz。其中，433MHz 频段频点支持默认设置与直接设置两种模式，其余频段只支持直接设置模式。779MHz 频段仅在 20MHz 晶振模式下才支持。

3.3.1 频点范围

在 20/26MHz 两种晶振模式下各频段范围与配置详见表 3-5 与表 3-6。

20M 晶振模式				
VCO_HB_SEL	PLL_TRXLB_SEL			
	2'b00	2'b01	2'b10	2'b11
1'b0	700-874MHz	350-437MHz	235-291MHz	-
1'b1	856-1045MHz	428-522MHz	285-350MHz	-

表 3-5 20MHz 晶振下各频段范围与配置

26M 晶振模式				
VCO_HB_SEL	PLL_TRXLB_SEL			
	2'b00	2'b01	2'b10	2'b11
1'b0	806-874MHz	403-437MHz	270-291MHz	-
1'b1	856-1045MHz	428-522MHz	285-348MHz	-

表 3-6 26MHz 晶振下各频段范围与配置

3.3.2 频点默认设置模式

HW3000 芯片的频点默认配置仅支持 433MHz 频段，频点配置由频点起始值、信道间隔和信道号三种配置组成。

频点起始值由寄存器 rf_freq_base（0x2F）配置，信道间隔由寄存器 ch_space（0x2F）配置，信道号由寄存器 channel1~channel8（0x28~0x2B）配置。其中 channel1 为主信道号，

channel2~channel8 为从信道号，从信道号只在跳频应用中使用，在没有跳频的情况下，channel1 的配置为当前信道号。最终发送或接收的主信道频点值为：

(RF_FREQ_BASE + CHANNEL1*2^{CH_SPACE})*100(KHz)。

频点起始值 rf_freq_base 寄存器设置值以 100KHz 为单位，即起始频点为：rf_freq_base*100(KHz)。例如配置起始频点为 433MHz，rf_freq_base 需设置值为 0x10EA，即十进制数 4330。

信道号取值范围是 0~255，即在单个应用中最多支持 256 个信道。

信道间隔有四种配置可选，具体如表 3-7 所示：

ch_space	信道间隔
00	100KHz
01	200KHz
10	400KHz
11	800KHz

表 3-7 信道间隔

3.3.3 频点直接设置模式

频点直接设置模式下，在频段范围内，用户可任意配置需要的频点。通过把 0x30 寄存器的 RF_PLL_DIRECT 位配置为 1，使能直接设置模式。

此模式可通过软件设置 Sigma-Delta 调制器的整数部分（Integer）和小数部分（Fraction）分频比。

频点的整数部分可通过寄存器 RF_FREQ_BASE 来配置，公式如下：

RF_FREQ_BASE = floor(f_c*k_{freq}/f_{xtal})。

频点的小数部分可通过 RF_FREQ_FRACTION 寄存器（0x30、0x31）来配置，公式如下：

RF_FREQ_FRACTION = round((f_c * k_{freq} / f_{xtal} -Integer)*2^{21})。

其中，f_c 为需要设置的信号载波频率，f_{xtal} 为使用晶振的频率，系数 k_{freq} 配置如表 3-8 所示，PLL_TRXLB_SEL 的取值详见表 3-5 与表 3-6。

k_{freq} 值配置	
PLL_TRXLB_SEL	k_{freq}
2’b00	2
2’b01	4
2’b10	6

表 3-8 k_{freq} 配置

如表 3-9 和表 3-10 所示，给出 20MHz 晶振与 26MHz 晶振下，典型频段的直接频点配置方式的寄存器配置值，可供用户参考。

频段（MHz）	寄存器配置（Hex）(@20MHz)					
	0x2C (BANK0)	0x35 (BANK0)	0x30 (BANK0)	0x31 (BANK0)	0x2F (BANK0)	0x17 (BANK1)
315	007B	5312	8010	0000	205E	F223
433	0052	3312	8013	3333	2056	F6C2
470	0052	3312	8000	0000	205E	F6C2
779	0029	0312	801C	CCCC	204D	FB61
868	0029	1312	8019	9999	2056	FB61
915	0029	1312	8010	0000	205B	FB61

表 3-9　20MHz 晶振各频段寄存器配置说明

频段（MHz）	寄存器配置（Hex）(@26MHz)					
	0x2C (BANK0)	0x35 (BANK0)	0x30 (BANK0)	0x31 (BANK0)	0x2F (BANK0)	0x17 (BANK1)
315	005F	5312	8016	2762	2048	F556
433	003F	3312	8013	B13B	2042	F8E3
470	003F	3312	8009	D89E	2048	F8E3
868	0020	1312	8018	9D89	2042	FC72
915	0020	1312	800C	4EC4	2046	FC72

表 3-10　26MHz 晶振各频段寄存器配置说明

3.3.4　频点配置注意事项

- 直接频点设置模式需要遵循一定的顺序，首先使能 rf_pll_direct，然后配置小数部分寄存器 rf_freq_fraction，最后配置整数部分寄存器 rf_freq_base。
- 若使用频点直接设置模式来配置频点，那么在软复位 soft reset、sleep、deep sleep 状态唤醒后，必须再重新配置一遍频点，否则频点恢复成默认值，导致收、发两端的频点不一致，通信失败。

3.4　晶振配置

HW3000 支持 20MHz 和 26MHz 两种晶振。如 3.2 章节所述配置均为 20MHz 晶振下的配置。

若芯片使用 26MHz 晶振，需设置 0x25 寄存器（BANK0）为 0x1201，且各频点配置值也会相应改变。另外，需要根据频段，更改 DEVIATION 寄存器设置值和 0x17（BANK1）寄存器值，具体配置方式可参考《HW3000_Datasheet_C》的 7.2.2 章节相关说明及表 3-9 和表 3-10 的配置。

3.5　发射功率配置

下图为芯片发射输出功率下的寄存器配置。

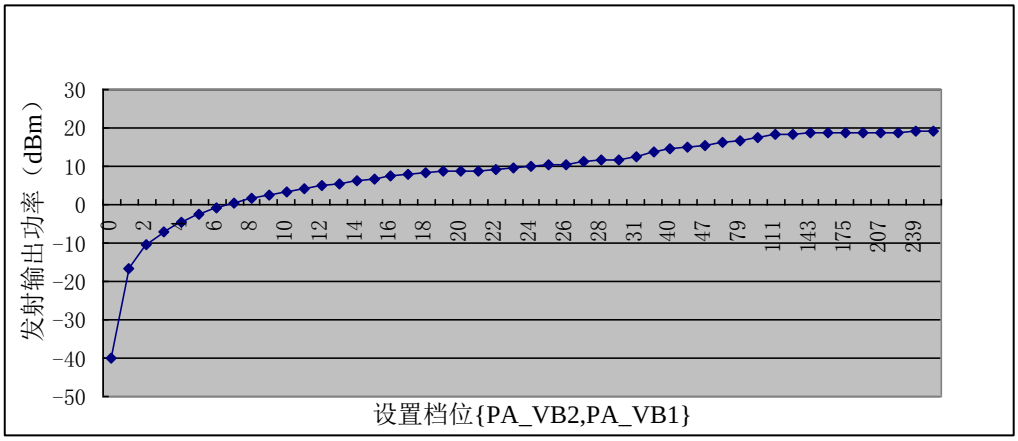

图 3-1　发送输出功率与配置关系示意图

下图为室温下，VDD = 3.0V 的芯片发射输出功率测试曲线图（横坐标为 0x40[15:8]寄存器设置值，纵坐标为输出功率），

发送功率 (dBm)	寄存器配置(Hex)	
	PA_VB2	PA_VB1
-40	0	0
-16	0	1
-10	0	2
-5	0	4
0	0	7
+5	0	C
+10	1	8
+15	2	A
+18	6	4
+20	F	F

表 3-11 不同发送功率下寄存器配置参考值

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。

第 4 章　注意事项

4.1　MISO 引脚无高阻态

HW3000 的 MISO 引脚无高阻态，所以在用到 MISO 引脚时，即 SPI 读的情况下，HW3000 不能跟其他芯片共用同一个 SPI。

4.2　AFC 和晶振校准

HW3000 在接收端提供载波频偏自动补偿功能（AFC），可通过 AFC_EN（0x25）寄存器使能晶振校准寄存器为 XOSC_CAL(0x37)，设置值支持 0x00 至 0xFF，步长约 15Hz。

4.3　数据 FIFO 深度

HW3000 收发数据 FIFO 深度为 256 字节，若用户采用增强型帧结构，用户可用的 FIFO 深度为 252 字节，若用户采用直接 FIFO 模式，通过半空半满中断可支持用户超过 256 字节的数据传输。

4.4　数据 FIFO 驻留

利用 HW3000 数据 FIFO 驻留的特点，可以提高效率。例如，若发射相同数据，可采用仅写入 FIFO 一次并通过清除 INT 中断的方式实现循环发送。利用该方式可更好地实现有源 RFID 功能。注，若存在接收数据，发射端 FIFO 将不再保留。

4.5　空中数据流方向

HW3000 空中数据流方向是 LSB first，比如 HW3000 的同步字设为 0xF398，二进制为 1111 0011 1001 1000，但因为是 LSB first，所以输出比特顺序为 0001 1001 1100 1111。

4.6　收发频偏

参照当前 HW3000 的初始化配置，在 10Kbps 速率下，若收、发两端的频偏大于 10KHz，则接收端基本接收不到数据。其原因是 10Kbps 为常用速率，为了提高 10Kbps 速率下的灵敏度、C/I 等性能，推荐的初始化代码在接收端滤波器等参数上进行了优化，滤波器带宽配置较窄。而其他速率如 19.2Kbps、38.4Kbps、50Kbps、100Kbps 等都是采用自动控制，允许的收发频偏较大。

因此，建议用户如果使用 10Kbps 速率，推荐采用高精度的晶振（≤10ppm），并且能获得较佳的通讯距离。

若用户对通讯距离要求不高，则可以选择滤波器参数自动控制，此方式下允许的收发频偏≤±17KHz。另外，还可以通过更改 BANK1 中 0x14 寄存器的【6:5】两位来对收发频偏进行调整：当前 0x14 寄存器配置值为 0x1935，若需要加大收发频偏，可改为 0x1915。

4.7　RSSI 读取

在接收模式时，芯片会评估天线端接收信号能量的大小，该数值会保存在寄存器 RSSI（0x23）中。RSSI 的读数单位为 dBm，数据的格式为二进制补码形式的符号数。

在 RSSI 寄存器里提供两个 RSSI 读数值，其中 RSSI1 保存的是上一个有效数据包（SFD 正确同步）的 RSSI 计算值，而 RSSI2 中保存的是实时的 RSSI 计算值，可用于 CSMA/CA 工作。

若需检测环境能量 RSSI2，建议在 RX 接收使能后至少延时 350us，再进行读取。另外如果用户系统 SPI 速率较快，大于 1Mbps 以上，在读取环境能量前，如有接收使能的关闭、打开操作，在接收使能切换中间必须有一定的延时操作，否则读取的 RSSI2 值会有跳变现象；同理，如果在读取 RSSI2 操作前，有发射、接收使能的切换操作，在状态切换中间必须加一定的延时操作。

在直接 FIFO 模式下，若读取上一个有效数据包的 RSSI1 值，必须在 SFD_INT 与 FIFO_INT 之间读取，即在同步之后，数据接收完成之前读取。数据接收完成之后，RSSI1 被复位成 0x81。

在增强型模式下，RSSI1 值会被锁定，直到下一个有效数据包之后才会被更新，且只有在芯片复位之后，RSSI1 值才会被复位成 0x81。

RSSI 与输入能量之间的关系示意图如图 4-1 所示，具体操作步骤如下：

(1) 芯片上电，初始化 HW3000 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，当 SFD 正确同步后（SFDDET_INT = ‘1’），读取 RSSI1(0x23，补码形式)。若需检测环境能量，建议在 RX 接收使能后延时 350us 左右读取 RSSI2。

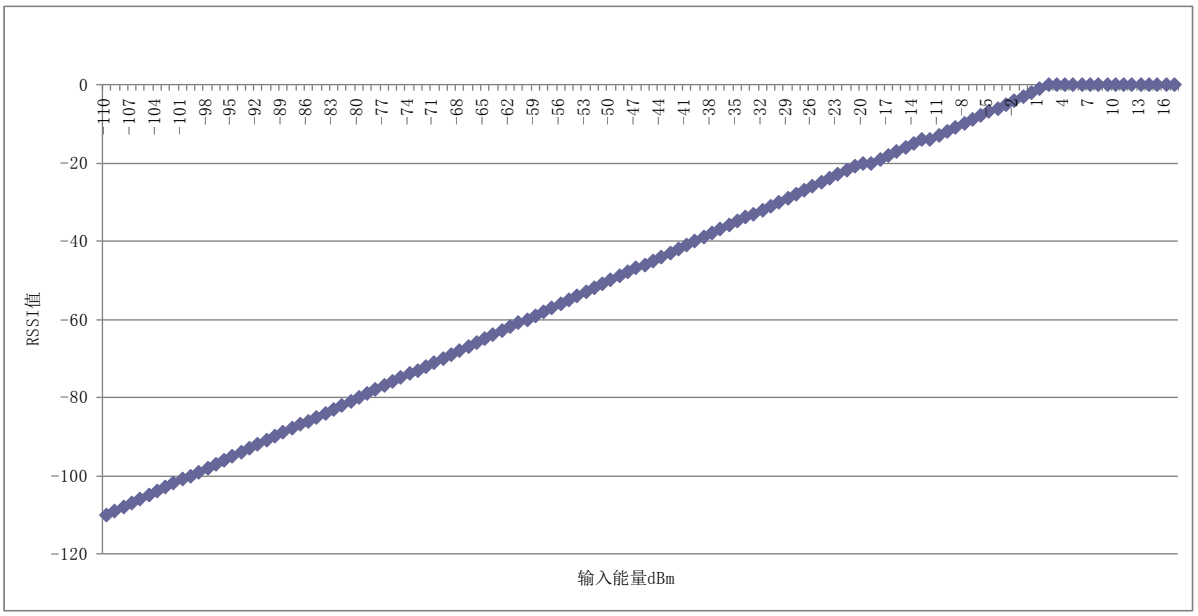

图 4-1　RSSI 与输入能量对应关系示意图

4.8　状态查询模式

当通过 SPI 轮询方式检测 HW3000 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。

4.9　PDN 状态的处理

HW3000 在进入 PDN（Power Down）状态后，HW3000 的 IRQ 引脚为低电平，而与 IRQ 连接的 MCU 引脚通常被配置为低电平触发中断。所以，在低功耗的应用里，当 HW3000 配置为

PDN 状态，若 MCU 也休眠，则在 MCU 休眠后，会因为 HW3000 的 IRQ 脚为低电平产生中断信号，将 MCU 误唤醒。

针对以上问题，处理方法有两个：

- 在 HW3000 进入 PDN 状态前，将 IRQ 脚的中断使能关闭，待 HW3000 从 PDN 状态唤醒后再将 IRQ 中断使能。
- 将 HW3000 的 IRQ 中断极性（0x1C 寄存器）修改为“由 0 变为 1”，即把 IRQ 配置为高电平中断。

4.10 复位功能区别

芯片从 PDN（Power Down）状态恢复至 IDLE 状态过程中，内部 POR 将复位全芯片(通过设置 PDN 输入引脚为高电平进入 PDN 状态)。

芯片从 DEEP SLEEP/SLEEP 恢复至 IDLE 状态，除配置寄存器外，所有寄存器都被复位（注意，寄存器中的状态指示位将被复位）。

芯片共提供两种软复位方式，分别为 SFT_RST0（0x60）和 SFT_RST1（0x61），其中：

- SFT_RST0 复位硬件电路与 FIFO，保留原有的寄存器设置值。
- SFT_RST1 进行全芯片复位，寄存器也会被复位成默认值。

4.11 状态切换说明

HW3000 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：

- HW3000 从 TX 状态转换到 RX 状态：首先关闭发射状态（包括清 FIFO_INT 中断标志，关闭发射使能 DBUS_TXEN），然后添加 SFT_RST0（0x60）软复位操作，注意如果频点配置使用频点直接设置模式，软复位后需要重新配置频点，其他寄存器不需要重新配置。软复位操作后，打开接收使能，芯片进入接收状态。
- HW3000 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_RST0（0x60）软复位操作，注意如果频点配置使用频点直接设置模式，软复位后需要重新配置频点，其他寄存器不需要重新配置。软复位操作后，打开发射使能，芯片进入发射状态。
- HW3000 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->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_RST0（0x60）软复位操作，然后再打开发射使能，发射下一包数据。
- HW3000 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->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_RST0（0x60）软复位操作，然后再打开 RX 使能，等待接收下一包数据。

4.12 低功耗说明

对于低功耗的应用系统，HW3000 不工作时也需要进入 POWER DOWN 状态。

HW3000 与 MCU 可能的交互管脚主要有以下接口：SPI 相关接口 CSN、SCK、MOSI、MISO；POWER DOWN 使能输入引脚 PDN；中断输出引脚 IRQ；通用数字 I/O 脚 GPIO0、GPIO1、GPIO2、GPIO3。

如果以上几个管脚与 MCU 的 IO 口相连，只要在初始化时将各个管脚做好初始化配置，如 CSN、SCK、MOSI、PDN 在 MCU 端配置为数字输出管脚，PDN 置低电平（PDN 置高电平时，HW3000 进入 POWER DOWN 状态），CSN 在没有 SPI 操作时置高电平。

MISO 和 IRQ 在 MCU 端配置为数字输入管脚。另外 SPI 管脚（SCK、MOSI、MISO）做 SPI 的相关配置，IRQ 做端口中断配置。

GPIO 的 4 个管脚如果与 MCU 的 I/O 管脚相连，根据 GPIO 配置寄存器（0x1D、0x1E）配置的 GPIO 功能，来配置 MCU 端的输入或输出特性，或者中断使能。

如果以上初始化配置完成，那么在休眠前 HW3000 与 MCU 交互的端口不用做特别处理，直接将 PDN 引脚置高电平，HW3000 即可正常进入 POWER DOWN 休眠状态，休眠功耗约 100nA。

如果 IRQ 没有与 MCU 的 IO 口相连，IRQ 脚在休眠前不必做任何处理。

如果 PDN 管脚直接接到地，HW3000 不能硬件复位，也就是不能进入到 POWER DOWN 休眠状态。

如果 GPIO 管脚没有与 MCU 的 I/O 管脚相连，GPIO 配置寄存器（0x1D、0x1E）对应的 GPIO 也不需要配置，休眠前也无需做特殊处理。

用户可以通过测量 VDD_RF 和 VDD_DIG 两个电源管脚的电流来判断 HW3000 是否进入 POWER DOWN 状态，两个电源管脚的电流和约 100nA，说明 HW3000 进入 POWER DOWN 状态。如果用户系统休眠电流偏大，用此方法也可以让用户快速的定位是 MCU 漏电还是 RF 漏电。

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，参见 HW3000 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：hw3000_write_reg
* 功能描述：写 HW3000 寄存器
* 输入参数：addr   寄存器地址
            value  寄存器值
* 返回参数：无
*****************************************************************************/
void hw3000_write_reg(uint8_t addr, uint16_t value)

/*****************************************************************************
* 函数名称：hw3000_read_reg
* 功能描述：读 HW3000 寄存器
* 输入参数：addr   寄存器地址
* 返回参数：value  寄存器值
*****************************************************************************/
uint16_t hw3000_read_reg(uint8_t addr)

/*****************************************************************************
* 函数名称：hw3000_write_fifo
* 功能描述：写 HW3000 FIFO
* 输入参数：addr    FIFO 地址
            data    数据地址
            length  数据长度
*****************************************************************************/
void hw3000_write_fifo(uint8_t addr, uint8_t *data, uint8_t length)
```

```
/*****************************************************************************
* 函数名称：hw3000_read_fifo
* 功能描述：写 HW3000 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 hw3000_read_fifo(uint8_t addr, uint8_t *data, uint8_t length)
```

5.3 SPI 汇编驱动例程

用户若采用上海东软载波微电子有限公司 HR7P 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 BCSN      0x00   //MCU 端口比特位选择，用户需要初始化输入/输出属性
#define BSCK      0x02
#define BMOSI     0x01
#define BMISO     0x03

#define TCSN      PBT0   //方向寄存器
#define TSCK      PAT2
#define TMOSI     PBT1
#define TMISO     PAT3

#define CSN       PB0
#define SCK       PA2
#define MOSI      PB1
#define MISO      PA3
```

```
void hw3000_write_reg(uint8_t addr, uint16_t value)
{
uint8_t i;
__asm
{
    MOVI    0x80
    IOR     (&addr&) % 0x80, 1
    BCC     PCSN, BCSN

    CLR     (&i&)   % 0x80

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

    CLR     (&i&)   % 0x80

    JBC     (&value&)   % 0x80 + 1, 7 ;write     value_h
    BSS     PMOSI, BMOSI
    JBS     (&value&)   % 0x80 + 1, 7
```

```
        BCC     PMOSI, BMOSI
        RL      (&value&)   % 0x80 + 1, 1
        BSS     PSCK, BSCK
        INC     (&i&)   % 0x80, 1
        BCC     PSCK, BSCK
        MOVI    0x08
        XOR     (&i&)   % 0x80, 0
        JBS     PSW, Z
        GOTO    $-0x0B

        CLR     (&i&)   % 0x80

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

        BSS     PCSN, BCSN
    }
    }

    uint16_t hw3000_read_reg(uint8_t addr)
```

```
{
    uint8_t i;
    uint16_t value;

__asm
{
    BCC     PCSN, BCSN

    CLR     (&i&)   % 0x80

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

    CLR     (&i&)   % 0x80

    BSS     PSCK, BSCK        ;read value
    BCC     PSW, C
    RL      (&value&)   % 0x80
    RL      (&value&)   % 0x80 + 1
```

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

        BSS     PCSN, BCSN
    }

    return value;
}

void hw3000_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, BCSN

        CLR     (&i&)   % 0x80

        JBC     (&addr&)   % 0x80, 7 ;write     addr
```

```
        BSS     PMOSI, BMOSI
        JBS     (&addr&)  % 0x80, 7
        BCC     PMOSI, BMOSI
        RL      (&addr&)  % 0x80, 1
        BSS     PSCK, BSCK
        INC     (&i&)  % 0x80, 1
        BCC     PSCK, BSCK
        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, BMOSI
            JBS     (&value&)  % 0x80, 7
            BCC     PMOSI, BMOSI
            RL      (&value&)  % 0x80, 1
            BSS     PSCK, BSCK
            INC     (&i&)  % 0x80, 1
            BCC     PSCK, BSCK
            MOVI    0x08
```

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

    __asm
    {
        BSS     PCSN, BCSN
    }
    }

    void hw3000_read_fifo(uint8_t addr, uint8_t *data, uint8_t length)
    {
    uint8_t i, j;
    uint8_t   value;

    __asm
    {
        BCC     PCSN, BCSN

        CLR     (&i&)   % 0x80

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

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

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

            BSS     PSCK, BSCK              ;read data
            BCC     PSW, C
            RL      (&value&)   % 0x80, 1
            BCC     PSCK, BSCK
            JBC     PMISO, BMISO
            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, BCSN
    }
    }
```

5.4 IO 模拟 SPI 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 hw3000_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 hw3000_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 hw3000_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 hw3000_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 数据发送例程

HW3000 支持增强型帧结构发射数据和直接 FIFO 模式发射数据，其中直接 FIFO 模式需要用户自行完成数据校验工作，直接 FIFO 模式可实现与市面同类无线芯片的互联互通。

```
int8_t hw3000_frame_tx(hw3000_mode_t hw3000_mode, hw3000_data_t *txbuf)
{
```

```
uint16_t reg_val;
if (txbuf->len > 252) {
    return -1;
}
if (_hw3000_state != TX) {
    _hw3000_state = TX;

    hw3000_write_reg(TRCTRL, 0x0100);     //tx_enable
    hw3000_write_reg(FIFOSTA, 0x0100); //flush fifo
    hw3000_write_fifo(FIFODATA, txbuf->data, txbuf->len); //write fifo
    hw3000_write_reg(FIFOCTRL, 0x0001); //ocpy = 1

    _hw3000_irq_request = 0;
    while (!_hw3000_irq_request); //wait for send finish
    _hw3000_irq_request = 0;
    _hw3000_state = IDLE;

    if (hw3000_mode.ack_mode == ENABLE) {
        reg_val = hw3000_read_reg(INTFLAG);
        if (reg_val & 0x0002) {
            hw3000_write_reg(FIFOCTRL, 0x0000);   //ocpy = 0
            hw3000_write_reg(INTIC, 0x0001);      //clr_int
            hw3000_write_reg(TRCTRL, 0x0000);     //send disable
            return -1;
        }
    }
    hw3000_write_reg(FIFOCTRL, 0x0000); //ocpy = 0
    hw3000_write_reg(INTIC, 0x0001);  //clr_int
    hw3000_write_reg(TRCTRL, 0x0000);     //send disable
```

```
            return 0;
        }
        return -1;
    }

    int8_t hw3000_fifo_tx(hw3000_mode_t hw3000_mode, hw3000_data_t *txbuf)
    {
        if (_hw3000_state != TX) {
            _hw3000_state = TX;
            hw3000_write_reg(TRCTRL, 0x0100);      //tx_enable
            hw3000_write_reg(FIFOSTA, 0x0108);    //flush fifo
            if (hw3000_mode.tx_mode == 0) {
                hw3000_write_reg(LEN0PKLEN, txbuf->len);
            }
            if (txbuf->len > 256) {
                hw3000_write_fifo(FIFODATA, txbuf->data, 256); //write fifo
                txbuf->len -= 256;
            } else {
                hw3000_write_fifo(FIFODATA, txbuf->data, txbuf->len); //write fifo
                txbuf->len = 0;
            }
            txbuf->idx = 0;
            hw3000_write_reg(FIFOCTRL, 0x0001);   //ocpy = 1
            _hw3000_irq_request = 0;
            _hw3000_gpio2_request = 0;
            while (1) {
                if (_hw3000_gpio2_request == 1 && txbuf->len != 0) {
                    _hw3000_gpio2_request = 0;
```

```
                    if (txbuf->len > 128) {
                        hw3000_write_fifo(FIFODATA,
&txbuf->data[256+128*txbuf->idx++], 128);      //write fifo
                        txbuf->len -= 128;
                    } else {
                        hw3000_write_fifo(FIFODATA,
&txbuf->data[256+128*txbuf->idx], txbuf->len); //write fifo
                        txbuf->len = 0;
                    }
                }
                if (_hw3000_irq_request == 1) {      //wait for send finish
                    _hw3000_irq_request = 0;
                    _hw3000_state = IDLE;
                    hw3000_write_reg(FIFOCTRL, 0x0000); //ocpy = 0
                    hw3000_write_reg(INTIC, 0x0001); //clr_int
                    hw3000_write_reg(TRCTRL, 0x0000); //send disable
                    return 0;
                }
            }
        }
        return -1;
    }
```

5.6 数据接收例程

HW3000 支持增强型帧结构发射数据和直接 FIFO 模式接收数据，其中直接 FIFO 模式需要用户自行完成数据校验工作，直接 FIFO 模式可实现与市面上大部分同类无线芯片互联互通。

```
int8_t hw3000_rx_enable(void)
{
    if (_hw3000_state != TX) {
        hw3000_write_reg(TRCTRL, 0x0080); //enable rx
```

```
            _hw3000_state = RX;
            _hw3000_irq_request = 0;
            _hw3000_gpio1_request = 0;
            return 0;
        }
        return -1;
    }

    void hw3000_rx_disable(void)
    {
        hw3000_write_reg(TRCTRL, 0x0000); //rx_disable
        hw3000_write_reg(INTIC, 0x0001); //clr_int
        _hw3000_state = IDLE;
        _hw3000_irq_request = 0;
        _hw3000_gpio1_request = 0;
    }

    int8_t hw3000_rx_data(hw3000_mode_t hw3000_mode, hw3000_data_t *rxbuf)
    {
        uint16_t reg;
        if (_hw3000_state == RX) {
            if (hw3000_mode.frame_mode == FRAME) {
                reg = hw3000_read_reg(FIFOCTRL);
                if (!(reg & 0xC000)) { //PHR CRC check
                    reg = hw3000_read_reg(RXPHR0);
                    rxbuf->len = ((reg >> 8) - 3) & 0x00FF;
                    hw3000_read_fifo(FIFODATA, rxbuf->data, rxbuf->len);
                    hw3000_write_reg(INTIC, 0x0001); //clr_int
                    return 0;
```

```
            }
        }
        else {     //derect FIFO mode
            hw3000_read_fifo(FIFODATA, rxbuf->data, HW3000_FULL_THRES);
             _hw3000_gpio1_request = 0;

            if (hw3000_mode.rx_mode == 1) {
                hw3000_write_reg(LEN0PKLEN, rxbuf->data[0]);
                 //FIFO mode1 (rx pcklenth: LEN0PKLEN)
            }
            rxbuf->len = rxbuf->data[0];
            rxbuf->len -= HW3000_FULL_THRES;
            rxbuf->idx = 1;

            while (1) {
                if (_hw3000_gpio1_request == 1) {
                    _hw3000_gpio1_request = 0;
                    hw3000_read_fifo(FIFODATA,
&rxbuf->data[HW3000_FULL_THRES * rxbuf->idx++], HW3000_FULL_THRES);
                    rxbuf->len -= HW3000_FULL_THRES;
                }
                if (_hw3000_irq_request == 1) {
                    _hw3000_irq_request = 0;

                    hw3000_read_fifo(FIFODATA,
&rxbuf->data[HW3000_FULL_THRES * rxbuf->idx], rxbuf->len);
                    rxbuf->len = 0;

                    break;
```

```
                    }
                }
                hw3000_write_reg(0x11, 0x0001); //clr_int
                return 0;
            }
            _hw3000_state = RX;
            _hw3000_gpio1_request = 0;
            _hw3000_irq_request = 0;
            hw3000_write_reg(0x11, 0x0001); //clr_int
            return -1;      //CRC error
        }
        else {
            return -1;
        }
    }
```

5.7 跳频接收例程

HW3000 支持硬件跳频接收功能（HOP_ENABLE = '1'，0x2D 寄存器）。在硬件跳频接收模式下，接收端将以设定的时间间隔扫描各个频点，当发现某一频点接收到有效信号后，接收会停留在该频点完成数据包的接收，系统最大可以扫描 8 个频点。

在频点默认设置模式下各信道频点：主信道频点+Δf1(CH_SPACE 设置的频点间隔)*CHANNELN。CH_SPACE 配置参考表 3-7。

在频点直接设置模式下各信道频点：主信道频点+Δf2(HOP_SPACE（0x3C、0x3D）设置的频点间隔)*CHANNELN，HOP_SPACE 具体的计算公式如下：

HOP_SPACE = round((fh* k_{freq} /fosc)*2^20)

其中 fh 为需要的扫描频率间隔，k_{freq} 取值详见表 3-8，fosc 为晶振频率。

（1）下面给出频点默认设置模式，433 频段，接收端在 4 个频点跳频接收的例程。

```
hw3000_write_reg(0x03, 0x4020);   //SFD 2bytes, preamble 32 bytes
hw3000_write_reg(0x24, 0x0206);   //default 0x0208   INV_PRE:6BYTES PRE:4BYTES
hw3000_write_reg(0x25, 0x5201);   //LP_TIMER:2*16
hw3000_write_reg(0x28, 0x0102);   //跳频信道号 01 跟 02
```

```
hw3000_write_reg(0x29, 0x0304);   //跳频信道号 03 跟 04
hw3000_write_reg(0x2D, 0x1447);   //HOP:4

_hw3000_rx_cnt1 = 0;
hw3000_rx_enable();
while(1){
    if((_hw3000_irq_request)||(_hw3000_gpio1_request)){
        if (hw3000_rx_data(_hw3000_mode, &_hw3000_data_buf) == 0){
            _hw3000_rx_cnt1++;
        }
    }
}
```

（2）下面给出频点直接设置模式，315 频段，接收端在 2 个频点跳频接收的例程。

```
hw3000_write_reg(0x30, 0x8011);
hw3000_write_reg(0x31, 0xEB85);
hw3000_write_reg(0x2F, 0x205E);    //315.2MHz

hw3000_write_reg(0x03, 0x4028);   //SFD 2bytes, preamble 40 bytes
hw3000_write_reg(0x24, 0x0410);   //INV_PRE:12BYTES PRE:8BYTES
hw3000_write_reg(0x25, 0x5201);   //LP_TIMER:2*16
hw3000_write_reg(0x28, 0x0002);   //跳频信道 315.2 和 315.2+2*0.1=315.4
hw3000_write_reg(0x2D, 0x0BA3);   //HOP:2
hw3000_write_reg(0x3D, 0x7AFD);   // HOP_SPACE 为 100kHz

_hw3000_rx_cnt1 = 0;
hw3000_rx_enable();
while(1){
    if((_hw3000_irq_request)||(_hw3000_gpio1_request)){
```

```
            if (hw3000_rx_data(_hw3000_mode, &_hw3000_data_buf) == 0){
                _hw3000_rx_cnt1++;
            }
        }
    }
```

（3）下面给出频点直接设置模式，433 频段，接收端在 2 个频点跳频接收的例程。

```
hw3000_write_reg(0x30, 0x8014);
hw3000_write_reg(0x31, 0x7AE1);
hw3000_write_reg(0x2F, 0x2056);     //433.2MHz

hw3000_write_reg(0x03, 0x4028);   //SFD 2bytes, preamble 40 bytes
hw3000_write_reg(0x24, 0x0410);   //INV_PRE:12BYTES PRE:8BYTES
hw3000_write_reg(0x25, 0x5201);   //LP_TIMER:2*16
hw3000_write_reg(0x28, 0x0002);   //跳频信道 433.2 和 433.2+2*0.1=433.4
hw3000_write_reg(0x2D, 0x0BA3);    //HOP:2
hw3000_write_reg(0x3D, 0x51EC);   // HOP_SPACE 为 100kHz

_hw3000_rx_cnt1 = 0;
hw3000_rx_enable();
while(1){
    if((_hw3000_irq_request)||(_hw3000_gpio1_request)){
        if (hw3000_rx_data(_hw3000_mode, &_hw3000_data_buf) == 0){
            _hw3000_rx_cnt1++;
        }
    }
}
```

（4）下面给出频点直接设置模式，915 频段，接收端在 2 个频点跳频接收的例程。

```
    hw3000_write_reg(0x30, 0x8010);
    hw3000_write_reg(0x31, 0xA3D7);
    hw3000_write_reg(0x2F, 0x205B);     //915.2MHz

    hw3000_write_reg(0x03, 0x4028);   //SFD 2bytes, preamble 40 bytes
    hw3000_write_reg(0x24, 0x0410);   //default 0x0208    INV_PRE:6BYTES PRE:4BYTES
    hw3000_write_reg(0x25, 0x5201);   //LP_TIMER:2*16
    hw3000_write_reg(0x28, 0x0002);   //跳频信道 915.2 和 915.2+2*0.1=915.4
    hw3000_write_reg(0x2D, 0x0BA3);    //HOP:2
    hw3000_write_reg(0x3D, 0x28F6);   // HOP_SPACE 为 100kHz

    _hw3000_rx_cnt1 = 0;
    hw3000_rx_enable();
    while(1){
        if((_hw3000_irq_request)||(_hw3000_gpio1_request)){
            if (hw3000_rx_data(_hw3000_mode, &_hw3000_data_buf) == 0){
                _hw3000_rx_cnt1++;
             }
        }
    }
```

第 6 章　芯片测试

HW3000 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，是常用的灵敏度测试方法，通常使用一台矢量信号发生器，按照一定的流程即可方便测试。PER 即误包率测试，通过误包率同样可以测试芯片的灵敏度，同时测试了芯片的收包能力。本章节主要介绍 PER 灵敏度的测试方法。

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+ HW3000 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 100 个数据包，计算 HW3000 模块收包数。
- HW3000 能接收到 90 个以上数据包的条件下，信号发生器的最低发射功率就是 HW3000 的接收灵敏度。

◆ 测试步骤

- 用射频同轴线，把 HW3000 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 HW3000 进入接收状态。
- 打开信号发生器，选择相应的 HW3000 数据文件，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 100 个脉冲，激励信号发生器发射 100 个数据包。
- 记录 HW3000 的收包数，当收包数达到 90 以上，把信号发生器的最低发射功率作为 HW3000 的接收灵敏度。

◆ 信号发生器产生的数据包格式

信号发生器产生增强型帧格式的数据包，内容如下：

- 帧格式表

<table>
<tr><td>4字节</td><td>2字节</td><td>1字节</td><td>1字节</td><td>1字节</td><td>1字节</td><td>32字节</td><td>2字节</td></tr>
<tr><td>前导码</td><td>帧分隔符</td><td>帧长</td><td>信道索引</td><td>标准识别号</td><td>帧头校验码</td><td>物理层载荷</td><td>帧校验序列</td></tr>
<tr><td colspan="2">SHR</td><td colspan="4">PHR</td><td>PSDU</td><td>FCS</td></tr>
</table>

- 前导码，4 个字节：0x55 0x55 0x55 0x55
- 帧分隔符，2 个字节：0xF3 0x98
- PHR，4 个字节：0x23 0xAA 0x55 0xCD
- PSDU，32 个字节：

0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A

0x0B 0x0C 0x0D 0x0E 0x0F 0x10 0x11 0x12 0x13 0x14

0x15 0x16 0x17 0x18 0x19 0x1A 0x1B 0x1C 0x1D 0x1E

0x1F 0x20

- FCS，2 个字节：0x6F 0xDC

第 7 章　常见问题分析

当 HW3000 芯片出现异常，要按照一定的逻辑进行问题分析：首先定位问题点，然后基于问题点进行逐一排查，确定故障原因。

例如，当 HW3000 芯片不能进行正常的收发通讯，首先将问题芯片模块与正常芯片模块进行收发通讯测试，定位芯片是发射还是接收问题，再从硬件和软件两个方面分别进行排查。

常见的排查思路如下：

◆ **硬件排查**

(1) 芯片供电是否正常

上电后芯片默认进入 IDLE 模式，芯片 VDR 引脚输出电压 1.8V 左右。

(2) 晶振是否正常

上电后测试晶振的 XTALP 或 XTALN 引脚，观察晶振是否起振，振荡频率是否正确。若有异常，可能是晶振损坏或起振电容不匹配，需要进行更换。

(3) SPI 读写是否正常

对寄存器进行读写操作，检查 SPI 的驱动程序是否正确。如果寄存器写入值与读取值不一致，可以用示波器抓取 SPI 波形，检查 SPI 四根线的电平是否正常，波形时序是否与数据手册一致；还需要检查 SPI 的通讯速率是否小于芯片 SPI 的最大通讯速率。

(4) 通讯频段有无干扰

将 Sub-1GHz 天线连接到频谱仪，直接测试空间辐射的信号，若在通讯频点存在较大的空间干扰，建议避开此频点进行通讯。

(5) 发射频点是否锁定

使能单载波发送模式，观察发射频点是否锁定，与设置值是否一致。

(6) 接收频点是否锁定

判断接收本振是否锁定，需要使用频谱仪，推荐的仪器参数为： Span=250KHz，Ref Amplitude=-50dBm。芯片使能接收，如果接收频点一直停留在接收设定频点*8 的位置不跳动，说明已锁定。

(7) 收发频偏是否过大

测试收发两端芯片的频偏值是否小于 30KHz（测量值与理论值之间的频偏），频偏软件补偿方法详见 HW3000 芯片数据手册的“AFC 及晶振校准功能”章节。

(8) 晶振频偏调整

调整晶振外接的两个电容，如果调整以后的晶振频率值一致性仍很差，则需要更换晶振，提高晶振精度。

◆ **软件排查**

(1) 寄存器检查

检查芯片的寄存器初始化配置，看是否与参考例程一致。

(2) 流程检查

检查芯片的收发流程，看是否与参考例程一致。

下面，再给出几种常见问题的解析：

✧ **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 将无法正确锁定。

✧ **写指针在什么时候需要软件清'0'？**

芯片内状态机写 FIFO（只出现在 PRX 端）的指针与 SPI 写 FIFO（只出现在 PTX 端）的指针复用同一个指针。PTX 与 PRX 在正常收发不切换时，硬件在合适情况下自动清零写指针，无需软件参与。但在收发角色切换（PRX 切换为 PTX）导致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'。

✧ **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 帧，通讯成功。
(2) PTX 重传超时，通讯不成功。

PTX 中断置起后可以通过 FIFO0_MAX_RETX 标志位来区分两种情况。

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，PTX 在下一次发送帧时 PID 不累加。

✧ **为什么 PRX 接收 CRC 正确，但接收 FIFO 的读取值与 PTX 发送 FIFO 内填写值不一致？**

(1) PTX 在写 FIFO 时，SPI 可能受到干扰造成误写，硬件根据误写的 FIFO 值自动生成 CRC，PRX 收到误写的值。
(2) PRX 在读 FIFO 时，SPI 可能受到干扰造成误读。

✧ **为什么收发双方 PIPE Address 配置一致时，会出现 PRX 的 PIPE 指示位（PRX_FIFO0_PIPE）所指示的 PIPE 与发送使用的 PIPE 不一致的现象？**

各 PIPE Address（0x14~0x1A）设置值之间的码间距，需大于接收同步字允许错误个数阈值 SYNC_THRES，否则接收各 PIPE 容易出现误同步。