移植指南

N32G45x系列到N32H47x_48x系列移植指南

简介

主要描述N32G45x系列与N32H47x_48x系列的资源、硬件及软件差异，便于快速从N32G45x系列切换到N32H47x_48x系列进行开发。

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

目录

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

1 简述

N32G45x系列是国民技术基于Cortex-M4内核的通用MCU产品，N32H47x及N32H48x系列是国民技术最新基于Cortex-M4内核开发的高性能通用MCU产品。

N32H47x及N32H48x系列相对于N32G45x系列新增了部分模块，且对部分模块设计进行了优化修改，新增了功能。

本指南主要介绍两个系列芯片的软件差异，内容如下：

1. 资源对比，主要展示N32G45x、N32H47x和N32H48x之间全系列的资源对比
2. 硬件对比，主要展示硬件设计方面的差异
3. 软件库，主要介绍官方开发套件SDK软件库文件结构及内容概述；
4. 模块总览，主要概述N32G45x系列和N32H47x_48x系列各模块的差异情况；
5. 软件差异
    a) 驱动API，详细对比各模块驱动API的函数及输入参数差异；
    b) 应用例程，介绍各模块在应用上的使用差异及注意点；

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

2 资源对比

注意点如下：

1. N32G455CEQ7 不支持USBFSD
2. N32G452系列ADC只支持2个，N32G455和N32G457系列支持4个
3. N32G452和N32G455系列不支持DVP和ETH模块

2.1 N32G45x与N32H47x对比

器件型号		N32G452CB/C/E N32G455CB/C/E			N32G452RB/C/E N32G457RC/E N32G455RB/C/E			N32G452MB/C/E N32G455MB/C/E N32G457MC/E			N32G452VB/C/E N32G455VB/C/E N32G457VC/E			N32G452QC/E N32G457QE		N32H473KCU7/8 N32H473KEU7/8		N32H473CCU7/8 N32H473CEU7/8 N32H474CCU7/8 N32H474CEU7/8 N32H473CCL7/8 N32H473CEL7/8 N32H474RCL7/8 N32H474REL7/8		N32H473RCL7/8 N32H473REL7/8 N32H474RCL7/8 N32H474REL7/8		N32H473MCL7/8 N32H473MEL7/8 N32H474MCL7/8 N32H474MEL7/8		N32H473VCL7/8 N32H473VEL7/8 N32H474VCL7/8 N32H474VEL7/8		N32H473QCL7/8 N32H473QEL7/8 N32H474QCL7/8 N32H474QEL7/8	
Flash容量（KB）		128	256	512	128	256	512	128	256	512	128	256	512	256	512	256	512	256	512	256	512	256	512	256	512	256	512	256	512
General SRAM容量（KB）		80	144	144	80	144	144	80	144	144	80	144	144	144	144	112	160	112	160	112	160	112	160	112	160	112	160	112	160
CCM SRAM容量（KB）		0														32													
Backup SRAM容量（KB）		0														4													
CPU频率		ARM Cortex-M4F @144MHz,180DMIPS														1.8~3.6V/-40~105℃ /125℃													
工作环境		1.8~3.6V/-40~105℃														ARM Cortex-M4F @200MHz, 250DMIPS													
定时器	通用	TIM2 TIM3 TIM4 TIM5														GTIM1 GTIM2 GTIM3 GTIM4 GTIM5 GTIM6 GTIM7 GTIM8 GTIM9 GTIM10														
	高级	TIM1 TIM8														ATIM1 ATIM2 ATIM3													
	基本	TIM6 TIM7														BTIM1 BTIM2													
	低功耗	NO														LPTIM1 LPTIM2													
通讯接口	SPI	SPI1 SPI2 SPI3														SPI1 SPI2 SPI3 SPI6		SPI1 SPI2 SPI3 SPI4 SPI6				SPI1 SPI2 SPI3 SPI4 SPI5 SPI6							
	I^2S	I2S2 I2S3														I2S2 I2S3													
	XSPI	QSPI（4线）														XSPI（8线）													
	I^2C	I2C1 I2C2 I2C4		I2C1 I2C2 I2C3 I2C4												I2C1 I2C2 I2C3 I2C4													
	USART	USART1 USART2 USART3														USART1 USART2 USART3													
	UART	UART4 UART5 UART6		UART4 UART5 UART6 UART7												UART5 UART6 UART7 UART8													
	USB FS Device	0（1）/USBFSD		USBFSD												USBFSD													
	CAN	CAN1 CAN2														FDCAN1 FDCAN2 FDCAN3(1)													
	SDIO	NO		SDIO												NO													
	DVP	0（3）/DVP														NO													
	ETH	0（3）/ETH														NO													
内存扩展	FEMC	NO														FEMC													
GPIO		37	42	51			65			80			97			26		42/38(2)		52		66		86		107			
DMA		2														2													
Number of channels		16 Channel														16 Channel													
12bit ADC		2（2）/4														4													
Number of channels		10Channel（2）/16Channel			16Channel（2）/22Channel			16Channel（2）/33Channel			16Channel（2）/38Channel			18Channel（2）/40Channel		13Channel		21Channel/20Channel(2)		26Channel		38Channel		45Channel		51Channel			
12bit DAC		2														8													
Number of channels		2Channel														8 (4 External/Internal + 4 Internal)													
OPAMP/PGA		OPAMP1 OPAMP2 OPAMP3 OPAMP4														PGA1 PGA2 PGA3 PGA4													
COMP		COMP1 COMP2 COMP3 COMP4		COMP1 COMP2 COMP3 COMP4 COMP5 COMP6 COMP7												COMP1 COMP2 COMP3 COMP4 COMP5 COMP6 COMP7													
算法支持		DES/3DES、AES、 SHA1/SHA224/SHA256、SM1、SM3、SM4F、SM7、MD5、CRC16/CRC32、TRNG														DES/3DES、AES、 SHA1/SHA224/SHA256、SM3、SM4、MD5、CRC16/CRC32、TRNG													
Cordic		NO														Yes													
FMAC		NO														Yes													
安全保护		读写保护（RDP/WRP）、存储加密、分区保护、安全启动														读写保护（RDP/WRP）、存储加密、分区保护、安全启动													
封装		LQFP48 QFN48		LQFP64			LQFP80			LQFP100			LQFP128			UQFN32		UQFN48 LQFP48		LQFP64		LQFP80		LQFP100		LQFP128			

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

2.1 N32G45x与N32H48x对比

器件型号		N32G452CB/C/E N32G455CB/C/E			N32G452RB/C/E N32G457RC/E N32G455RB/C/E			N32G452MB/C/E N32G455MB/C/E N32G457MC/E			N32G452VB/C/E N32G455VB/C/E N32G457VC/E			N32G452QC/E N32G457QE		N32H482REL7 N32H487REL7	N32H482VEL7 N32H487VEL7	N32H482ZEL7 N32H487ZEL7
Flash容量（KB）		128	256	512	128	256	512	128	256	512	128	256	512	256	512	512	512	512
General SRAM容量（KB）		80	144	144	80	144	144	80	144	144	80	144	144	144	144	160	160	160
CCM SRAM容量（KB）		0														32		
Backup SRAM容量（KB）		0														4		
CPU频率		ARM Cortex-M4F @144MHz,180DMIPS														ARM Cortex-M4F @240MHz, 300DMIPS		
工作环境		1.8~3.6V/-40~105℃														1.8~3.6V/-40~105℃		
定时器	通用	TIM2 TIM3 TIM4 TIM5														GTIM1 GTIM2 GTIM3 GTIM4 GTIM5 GTIM6 GTIM7 GTIM8 GTIM9 GTIM10		
	高级	TIM1 TIM8														ATIM1 ATIM2 ATIM3		
	基本	TIM6 TIM7														BTIM1 BTIM2		
	低功耗	NO														LPTIM1 LPTIM2		
通讯接口	SPI	SPI1 SPI2 SPI3														SPI1 SPI2 SPI3 SPI4 SPI6	SPI1 SPI2 SPI3 SPI4 SPI5 SPI6	
	I^2S	I2S2 I2S3														I2S2 I2S3		
	XSPI	QSPI（4线）														XSPI(8线)		
	I^2C	I2C1 I2C2 I2C4			I2C1 I2C2 I2C3 I2C4											I2C1 I2C2 I2C3 I2C4		
	USART	USART1 USART2 USART3														USART1 USART2 USART3 USART4		
	UART	UART4 UART5 UART6			UART4 UART5 UART6 UART7											UART5 UART6 UART7 UART8		
	USB FS Device	0（1）/USBFSD			USBFSD											USBFSD		
	USBHS Dual Role	NO														USBHS		
	CAN	CAN1 CAN2														FDCAN1 FDCAN2		
	SDIO	NO			SDIO											SDIO		
	DVP	0（3）/DVP														DVP		
	ETH	0（3）/ETH														ETH		
内存扩展	FEMC	NO														No	FEMC	
GPIO		37		42	51			65			80			97		54	85	118
DMA		2														2		
Number of channels		16 Channel														16Channel		
12bit ADC		2（2）/4														4	4	4
Number of channels		10Channel（2）/16Channel			16Channel（2）/22Channel			16Channel（2）/33 Channel			16Channel（2）/38Channel			18Channel(2) /40Channel		26Channel	42Channel	51Channel
12bit DAC		2														2		
Number of channels		2Channel														2 External		
OPAMP/PGA		OPAMP1 OPAMP2 OPAMP3 OPAMP4														NO		
COMP		COMP1 COMP2 COMP3 COMP4			COMP1 COMP2 COMP3 COMP4 COMP5 COMP6 COMP7											NO		
算法支持		DES/3DES、AES、 SHA1/SHA224/SHA256、SM1、SM3、SM4F、SM7、MD5、CRC16/CRC32、TRNG														DES/3DES、AES、 SHA1/SHA224/SHA256、SM3、SM4、MD5、CRC16/CRC32、TRNG		
Cordic		NO														Yes		
FMAC		NO														Yes		
安全保护		读写保护（RDP/WRP）、存储加密、分区保护、安全启动														读写保护（RDP/WRP）、存储加密、分区保护、安全启动		
封装		LQFP48 QFN48			LQFP64			LQFP80			LQFP100			LQFP128		LQFP64	LQFP100	LQFP144

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

2.1 N32H47x与N32H48x对比

器件型号		N32H482REL7 N32H487REL7	N32H482VEL7 N32H487VEL7	N32H482ZEL7 N32H487ZEL7	N32H473KCU7/8 N32H473KEU7/8		N32H473CCU7/8 N32H473CEU7/8		N32H473RCL7/8 N32H473REL7/8		N32H473MCL7/8 N32H473MEL7/8		N32H473VCL7/8 N32H473VEL7/8		N32H473QCL7/8 N32H473QEL7/8	
工作环境		1.8~3.6V/-40~105℃			1.8~3.6V/-40~105℃ /125℃											
CPU频率		ARM Cortex-M4F @240MHz, 300DMIPS			ARM Cortex-M4F @200MHz, 250DMIPS											
Flash容量（KB）		512	512	512	256	512	256	512	256	512	256	512	256	512	256	512
Total SRAM	General SRAM	160	160	160	112	160	112	160	112	160	112	160	112	160	112	160
(KB)	CCM SRAM	32			32											
	Backup SRAM	4			4											
定时器	ATIM	ATIM1 ATIM2 ATIM3			ATIM1 ATIM2 ATIM3											
	GTIM	GTIM1 GTIM2 GTIM3 GTIM4 GTIM5 GTIM6 GTIM7 GTIM8 GTIM9 GTIM10			GTIM1 GTIM2 GTIM3 GTIM4 GTIM5 GTIM6 GTIM7 GTIM8 GTIM9 GTIM10											
	BTIM	BTIM1 BTIM2			BTIM1 BTIM2											
	LPTIM	LPTIM1 LPTIM2			LPTIM1 LPTIM2											
通讯	SPI/I2S	SPI1 SPI2 SPI3 SPI4 SPI6	SPI1 SPI2 SPI3 SPI4 SPI5 SPI6		SPI1 SPI2 SPI3 SPI6		SPI1 SPI2 SPI3 SPI4 SPI6				SPI1 SPI2 SPI3 SPI4 SPI5 SPI6					
	I2S	I2S2 I2S3			I2S2 I2S3											
	I^2C	I2C1 I2C2 I2C3 I2C4			I2C1 I2C2 I2C3 I2C4											
	USART	USART1 USART2 USART3 USART4			USART1 USART2 USART3 USART4											
	UART	UART5 UART6 UART7 UART8			UART5 UART6 UART7 UART8											
	USB FS Device	USBFSD			USBFSD											
	USB HS DualRole	USBHS			NO											
	FDCAN	FDCAN1 FDCAN2			FDCAN1 FDCAN2 FDCAN3(1)											
	Ethernet	ETH			NO											
存储扩展	XSPI	XSPI			XSPI											
	FEMC	No	FEMC		No								FEMC			
	SDIO	SDIO			NO											
人机交互	DVP	DVP			NO											
GPIO		54	85	118	26		42/38(2)		52		66		86		107	
DMA		2			2											
Number of channels		16Channel			16 Channel											
12bit ADC		4	4	4	4		4		4		4		4		4	
Number of channels		26Channel	42Channel	51Channel	13Channel		21Channel/20Chan		26Channel		38Channel		45Channel		51Channel	
12bit DAC		2			8											
Number of channels		2 External			8 (4 External/Internal + 4 Internal)											
OPAMP/PGA		NO			PGA1 PGA2 PGA3 PGA4											
COMP		NO			COMP1 COMP2 COMP3 COMP4 COMP5 COMP6 COMP7											
算法支持		DES/3DES、AES、 SHA1/SHA224/SHA256、SM3、SM4、MD5、CRC16/CRC32、TRNG			DES/3DES、AES、 SHA1/SHA224/SHA256、SM3、SM4、MD5、CRC16/CRC32、TRNG											
Cordic		Yes			Yes											
FMAC		Yes			Yes											
安全保护		读写保护（RDP/WRP）、存储加密、分区保护、安全启动			读写保护（RDP/WRP）、存储加密、分区保护、安全启动											
封装		LQFP64	LQFP100	LQFP144	UQFN32		UQFN48 LQFP48		LQFP64		LQFP80		LQFP100		LQFP128	

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

3 硬件差异

3.1 供电方案

N32H47x/48x系列与N32G45x系列对比，设计差异主要为VREF+供电。

N32H47x/48x系列(VREF+参考电压)：

- 当VERF+使用内置参考源VREFBUF时，VREF+引脚建议就近放置一个0.1uF和一个1uF的电容。
- 当VERF+由外部供电时，VREF+引脚建议就近放置一个0.1uF和一个10uF的电容。

3.2 启动引脚

N32G45x系列：BOOT0、BOOT1.

N32H47x/48x系列：BOOT0.

3.3 ADC转换器

N32G45x系列：

N32G45x系列在ADC输入时钟为72MHZ条件下，ADC快速通道采样率不超过5Msps，ADC慢速通道采样率建议不超过2.5Msps.

N32H47x/48x系列：

N32H47x/N32H48x系列在ADC输入时钟为80MHZ条件下，ADC快速通道采样率不超过4.7Msps，ADC慢速通道采样率建议不超过2.5Msps.

3.4 IO耐压值

N32G45x系列：

FT：容忍5V；TTa：容忍3.3V，支持模拟外设；TC：普通3.3V I/O.

N32H47x/48x系列：

FT：容忍5V；FTa：容忍5V，支持模拟外设。

3.5 运算放大器

N32G45x系列：

内嵌最多4个独立的运算放大器（OPAMP），具有外部放大、内部跟随和可编程放大器（PGA）等多种工作模式(或兼具有内部放大和外部滤波)。

N32H47x系列：

N32H47x系列芯片内包含4个灵活的可编程增益放大器（PGA），输出可选择内部连接到ADC或者COMP的输入通道,不支持外部放大。详见对应数据手册的引脚复用定义。

N32H48x系列：

N32H48x系列芯片不支持运算放大器。

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

3.6 TSC触摸传感器

N32G45x系列：

最大支持24个电容式触控通道。

N32H47x/48x系列：

不支持TSC触控功能。

3.7 超高精度定时器(SHRTIM)

N32G45x系列：

不支持超高精度定时器。

N32H47x/48x系列：

N32H47x支持一个超高精度定时器。

N32H48x不支持超高精度定时器。

3.8 CAN

N32G45x系列：

支持2路CAN总线接口，兼容规范2.0A和2.0B(主动)。不支持全管脚映射。

N32H47x/48x系列：

最多支持3个FDCAN，支持CAN 2.0A/B与CAN FD协议，兼容非ISO标准的Bosch协议。支持全管脚映射。

3.9 U(S)ART

N32G45x系列：

支持3个通用同步/异步收发器(USART1、USART2和USART3)，和4个通用异步收发器(UART4、UART5、UART6、UART7)，不支持全管脚映射。

N32H47x/48x系列：

支持4个通用同步/异步收发器(USART1、USART2和USART3)，和4个通用异步收发器(UART4、UART5、UART6、UART7)，部分管脚支持全管脚映射。

3.10 I2S

N32G45x系列：

仅支持半双工通讯。

N32H47x/48x系列：

支持全双工通讯。

3.11 XSPI

N32G45x系列：

支持Single SPI、Dual SPI、Quard SPI模式。

N32H47x/48x系列：

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

支持Single SPI、DUAL SPI、QUAD SPI、Dual-QUAD、OCTAL SPI模式。

3.12 DVP

N32G45x系列：

最多支持14位的传统同步并行接口。

N32H47x/48x系列：

支持8位、10位、12位和16位可配置的传统同步并行接口。

3.13 USB_HS_DualRole

N32G45x系列:

不支持USB高速双角色接口（USB HS Dual Role）。

N32H47x/48x系列：

内嵌一个USB高速双角色接口（USB HS Dual Role），支持Host模式和Device模式。

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

4 软件差异

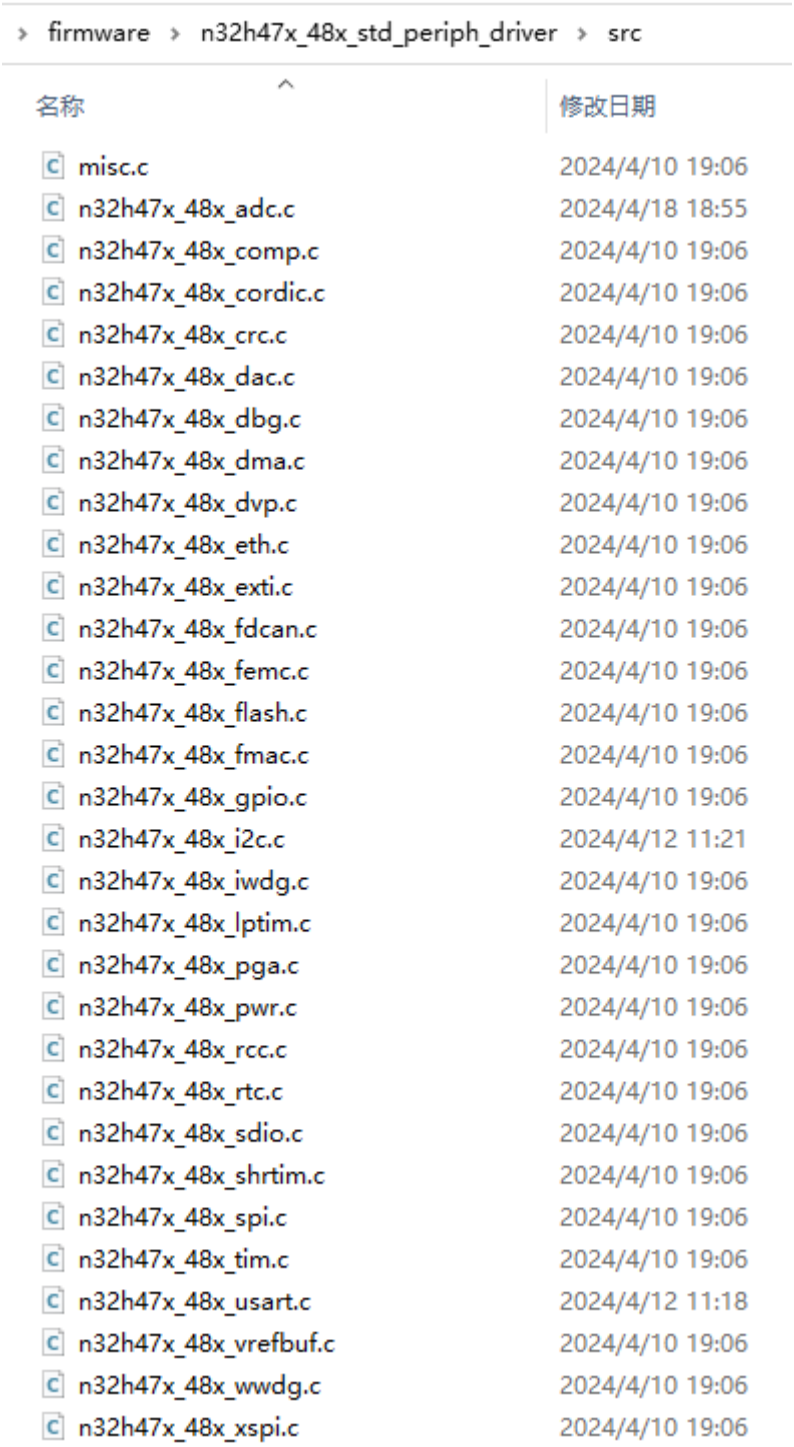
4.1 软件库介绍

国民技术会提供官方开发套件，该套件中包含了SDK软件库、数据手册、用户手册、使用指南、应用笔记等资料，其中SDK软件库文件结构如下：

文件夹	子文件夹1	子文件夹2	说明
firmware	CMSIS	core	内核文件：必须添加到应用工程
		device	启动文件及系统配置文件：必须添加到应用工程
	n32xxxx_std_periph_driver	inc	各模块标准固件库的.h和.c文件：根据实际应用情况选择模块驱动
		src	
	n32xxxx_algo_lib	inc	安全算法库：根据实际应用情况选择
		lib	
	n32xxxx_usbfs_driver	inc	USBFS库：根据实际应用情况选择
		lib	
	n32xxxx_usbhs_driver	device	USBHS库：根据实际应用情况选择 *N32G45x无此文件夹*
		driver	
		host	
middlewares	rt-thread	/	*N32H47x_48x无此文件夹*
	fat_fs	src	文件系统（SDK中用于USB例程）：根据实际应用情况选择 *N32G45x无此文件夹*
	FreeRTOSv202212.01	Source	FreeRTOS操作系统（SDK中用于以太网例程）：根据实际应用情况选择 *N32G45x无此文件夹*
	lwip-2.2.0	src	以太网lwip专用协议栈：根据实际应用情况选择 *N32G45x无此文件夹*
projects	n32xxxx_EVAL	bsp	基于开发板的打印及延时配置：根据实际应用情况选择
		examples	各模块的典型应用例程：根据实际应用情况选择模块例程参考

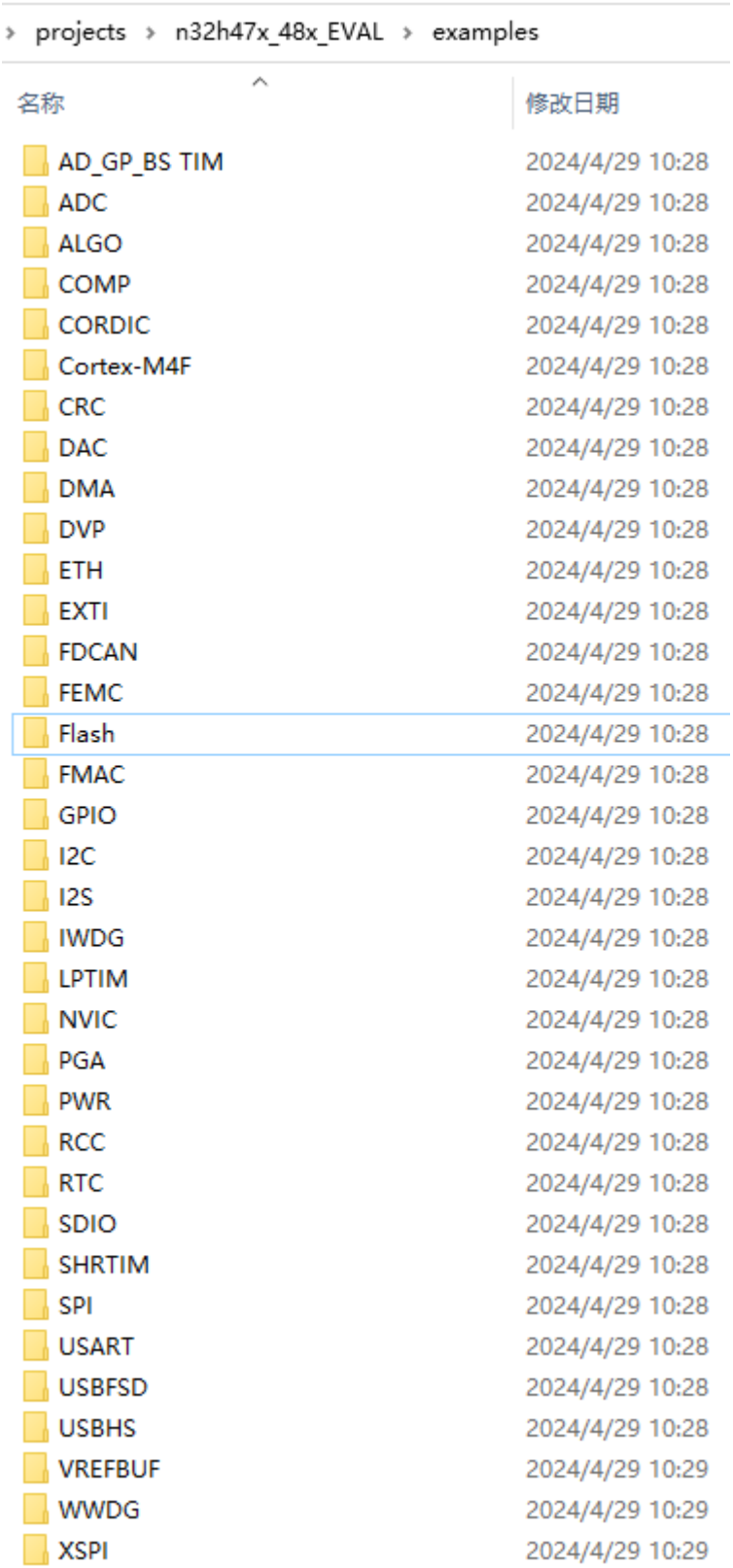
国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

各模块驱动（n32xxxx_std_periph_driver）：



国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

各模块例程（examples）：



4.2 模块总览

下表列出N32G45x系列和N32H47x_48x系列各模块的差异情况总览，在“模块驱动API详细对比及使用

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

注意点”章节，差异较小的模块会详细展示驱动API的函数及输入参数差异，差异较大或新增的模块难以详细对比，仅提供使用指导及注意点，建议参考SDK例程进行应用开发。

N32G45x系列	N32H47x_48x系列	差异概述
RCC	RCC	系统模块，差异较大
PWR	PWR	系统模块，差异较小，新增部分功能
FLASH	FLASH	FLASH模块，差异较小，新增部分功能
GPIO	GPIO	GPIO基本配置及重映射配置差异较大，并新增部分功能
MISC	MISC	系统模块，函数一致，可直接替换
ADC	ADC	差异较大，新增功能比较多
BKP	-	删除BKP模块
CAN	FDCAN	N32G45x系列只支持CAN 2.0A/B，N32H47x_48x系列支持FDCAN，差异较大
COMP	COMP	差异较小
CRC	CRC	差异较小
DAC	DAC	功能差异较大，新增功能较多
DBG	DBG	N32H47x_48x支持SHRTIM调试
DMA	DMA	差异较小，新增burst功能、部分通道请求映射有差异等
DVP	DVP	功能差异大
ETH	ETH	IP有升级，以太网功能不受影响，但驱动函数和示例工程代码均有较大差异
EXTI	EXTI	差异较小，外部中断线由22条增加至32条，且外部中断支持管脚全映射
I2C	I2C	N32G45x系列I2C没有FIFO收发功能功能，N32H47x_48x系列I2C支持FIFO功能
IWDG	IWDG	N32G45x不支持冻结，N32H47x_48x支持冻结
OPAMP	PGA	修改模块命名，功能差异比较大
QSPI	XSPI	修改模块命名，N32H47x_48x支持8线模式，例程差异较大
RTC	RTC	N32G45x无参考时钟输入，BKP和RTC是分开的两个模块，N32H47x_48x支持参考时钟输入，BKP和RTC合并成了一个模块，支持入侵中断连接到EXTI19
SDIO	SDIO	模块功能差异较小，部分寄存器位定义有调整，SDK新增部分接口API函数
SPI	SPI	N32G45x系列支持3个SPI接口（SPI1~SPI3），2个I2S接口和SPI2、SPI3复用，N32H47x_48x系列支持6个SPI接口（SP11~SPI6），2个I2S接口和SPI2、SPI3复用； N32G45x系列SPI不支持FIFO功能，N32H47x_48x系列SPI支持FIFO功能； N32G45x系列I2S音频采样频率配置范围从8KHz 到96KHz，N32H47x_48x系列I2S音频采样频率配置范围从8KHz到192KHz。
TIM	TIM	刹车增加一路，刹车源增加单独的使能和极性控制位，增加双向刹车IO的功能，刹车源增加。编码器模式增加。互联关系变化较大
TSC	-	删除TSC模块
USART	USART	差异较小，新增部分功能
WWDG	WWDG	N32G45x窗口值为7bit，N32H47x_48x为14bit
ALGO	ALGO	驱动封装成库，调用接口函数即可
USBFS	USBFSD	修改模块命名，N32H47x_48x支持无晶体模式
-	USBHS	新增USBHS模块
-	CORDIC	新增CORDIC模块

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

-	FEMC	新增FEMC模块
-	FMAC	新增FMAC模块
-	LPTIM	新增LPTIM模块
-	SHRTIM	新增SHRTIM模块
-	VREFBUF	新增VREFBUF模块

国民技术股份有限公司 Nations Technologies Inc.
地址：深圳市南山区高新北区宝深路 109 号国民技术大厦
电话：+86-755-86309900 传真：+86-755-86169100
邮箱：info@nationstech.com 邮编：518057

4.3 模块驱动API详细对比及使用注意点

4.3.1 RCC

时钟控制为系统模块，N32G45x系列和N32H47x_48x系列设计上差异较大，建议用户直接参考N32H47x_48x系列例程，在进入main之前会自动调用system_n32h47x_48x.c里的函数进行时钟初始化，用户通过如下宏定义选择不同的时钟源和系统时钟频率即可：

```
#define SYSCLK_USE_HSI     0
#define SYSCLK_USE_HSE     1
#define SYSCLK_USE_HSI_PLL 2
#define SYSCLK_USE_HSE_PLL 3

#ifndef SYSCLK_FREQ
#if defined (N32H473) || defined (N32H474)
#define SYSCLK_FREQ    200000000
#elif defined (N32H475) || defined (N32H482) || defined (N32H487)
#define SYSCLK_FREQ    240000000
#else
#error wrong macro definition
#endif

#endif

#ifndef SYSCLK_SRC
#define SYSCLK_SRC   SYSCLK_USE_HSI_PLL
#endif
```

根据芯片系列配置系统时钟频率

选择系统时钟源

同时RCC模块的“RCC_ClockConfig”示例工程代码也提供了其他系统时钟的配置可供参考，根据宏定义，系统时钟源还可以配置为SHRTPLL或USB240M：

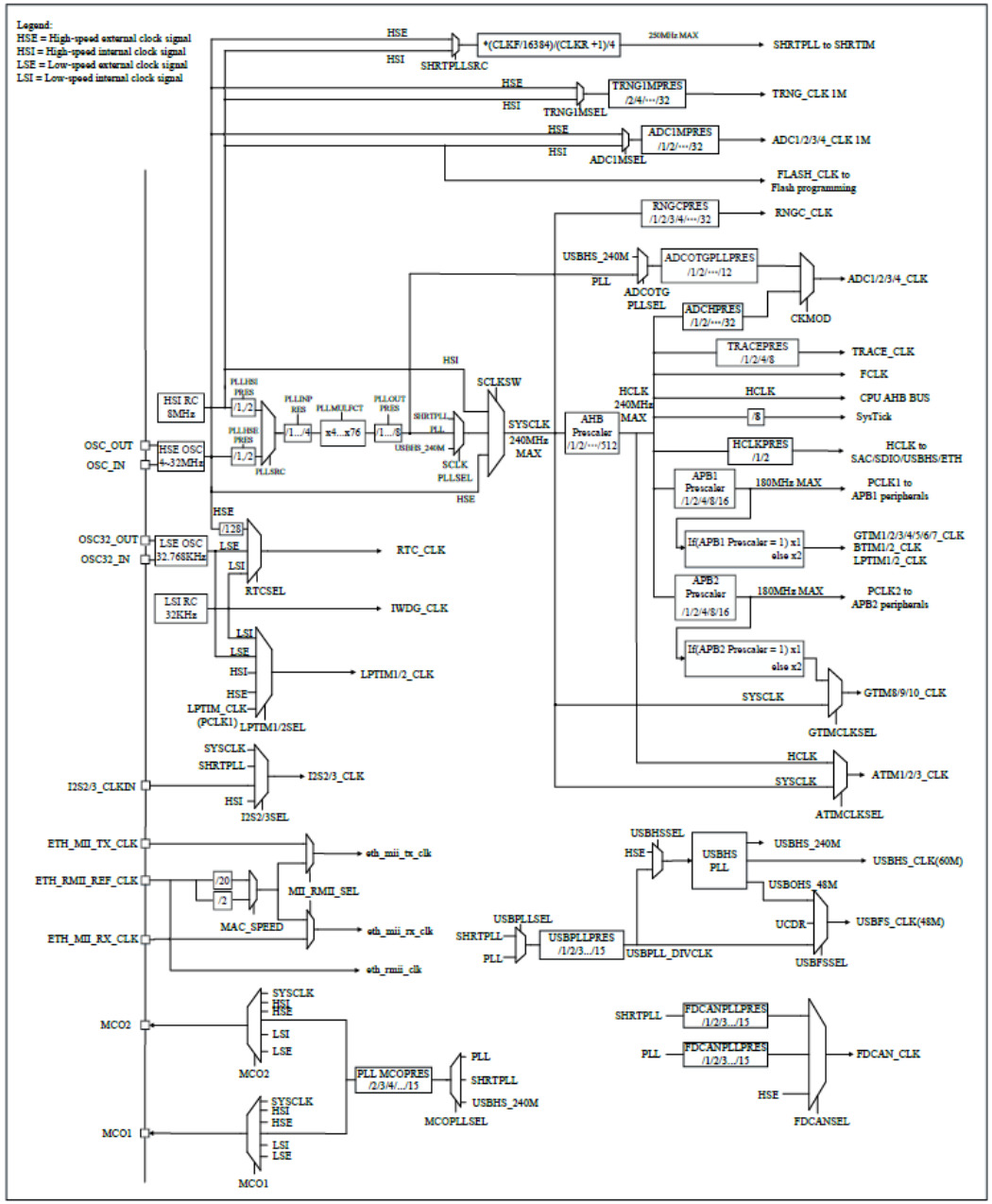
main.c

```c
{
    log_init(); /* should reinit after sysclk changed */
    log_info("--------------------------------\n");
    log_info("%s:\n", msg);
    RCC_GetClocksFreqValue(&RCC_ClockFreq);
    log_info("SYSCLK: %d\n", RCC_ClockFreq.SysclkFreq);
    log_info("HCLK: %d\n", RCC_ClockFreq.HclkFreq);
    log_info("PCLK1: %d\n", RCC_ClockFreq.Pclk1Freq);
    log_info("PCLK2: %d\n", RCC_ClockFreq.Pclk2Freq);
}

int main(void)
{

    PrintfClockInfo("After reset");

/*** Select one of the following configuration methods ***/
#if SYSCLK_SOURCE_SELECT == SYSCLK_SOURCE_HSI
  /* Method 1  */
    SetSysClockToHSI();
    PrintfClockInfo("HSI->SYSCLK, 8MHz");
#elif SYSCLK_SOURCE_SELECT == SYSCLK_SOURCE_HSE
  /* Method 2  */
    SetSysClockToHSE();
    PrintfClockInfo("HSE->SYSCLK, 8MHz");
#elif SYSCLK_SOURCE_SELECT == SYSCLK_SOURCE_PLL
    /* Method 3  */
    SetSysClockToPLL(RCC_PLL_SRC_HSE,160000000);
    PrintfClockInfo("HSE->PLL->SYSCLK, 160M");
#elif SYSCLK_SOURCE_SELECT == SYSCLK_SOURCE_SHRTPLL
  /* Method 4  */
    SetSysClockToSHRTPLL(RCC_SHRTPLL_SRC_HSI, HSI_VALUE, (240000000U));
    PrintfClockInfo("HSI->SHRTPLL->SYSCLK, 240M");
#elif SYSCLK_SOURCE_SELECT == SYSCLK_SOURCE_USBHS240M
  /* Method 5  */
    SetSysClockToUSBHS240M();
    PrintfClockInfo("HSE->SHRTPLL->USBHS240M->SYSCLK, 240M");
#endif

    /* Output HSE clock on MCO pin */
    RCC_EnableAHB1PeriphClk(RCC_AHB_PERIPHEN_GPIOA, ENABLE);
    RCC_EnableAPB2PeriphClk(RCC_APB2_PERIPH_AFIO, ENABLE);

    GPIO_InitStruct(&GPIO_InitStructure);
    GPIO_InitStructure.Pin             = GPIO_PIN_7;
    GPIO_InitStructure.GPIO_Mode       = GPIO_MODE_AF_PP;
    GPIO_InitStructure.GPIO_Alternate  = (uint32_t)13;
    GPIO_InitPeripheral(GPIOA, &GPIO_InitStructure);

    /* If the MCO selects PLL, SHRTPLL or USBHS240M, the prescaler can be configured first */
    RCC_ConfigMcoClkPre(RCC_MCO_PLLCLK_DIV10);
    /* Select the corresponding MCO clock source */
    RCC_ConfigMco1(RCC_MCO_SYSCLK);

    while (1);
}
```

n32h47x_48x_it.h

```c
*
*     NATIONS products and technologies shall not be used for or incorporated
* whose manufacture, use, or sale is prohibited under any applicable domestic
* User shall comply with any applicable export control laws and regulations p
* the governments of any countries asserting jurisdiction over the parties or
**/

/**
*\*\file n32h47x_48x_it.h
*\*\author Nations
*\*\version v1.0.0
*\*\copyright Copyright (c) 2023, Nations Technologies Inc. All rights reserv
**/
#ifndef __N32H47X_48X_IT_H__
#define __N32H47X_48X_IT_H__

#ifdef __cplusplus
extern "C" {
#endif

#include "n32h47x_48x.h"
#include "n32h47x_48x_rcc.h"
#include "n32h47x_48x_usart.h"
#include "n32h47x_48x_gpio.h"
#include "n32h47x_48x_flash.h"
#include "misc.h"

#define SYSCLK_SOURCE_HSI         0
#define SYSCLK_SOURCE_HSE         1
#define SYSCLK_SOURCE_PLL         2
#define SYSCLK_SOURCE_SHRTPLL     3
#define SYSCLK_SOURCE_USBHS240M   4


#ifndef SYSCLK_SOURCE_SELECT
#define SYSCLK_SOURCE_SELECT  SYSCLK_SOURCE_PLL /*select sysclk source */
#endif

void NMI_Handler(void);
void HardFault_Handler(void);
void MemManage_Handler(void);
void BusFault_Handler(void);
void UsageFault_Handler(void);
void SVC_Handler(void);
void DebugMon_Handler(void);
void PendSV_Handler(void);
void SysTick_Handler(void);
void RCC_IRQHandler(void);

#ifdef __cplusplus
}
#endif

#endif /* __N32H47X_48X_IT_H__ */
```

如果还有其他特殊应用要求，则用户可根据如下时钟树自行配置时钟，详细参考用户手册：


Legend:
HSE = High-speed external clock signal
HSI = High-speed internal clock signal
LSE = Low-speed external clock signal
LSI = Low-speed internal clock signal
HSE
HSI
SHRTPLLSRC
*(CLKF/16384)/(CLKR +1)/4
250MHz MAX
SHRTPLL to SHRTIM
TRNG1MSEL
TRNG1MPRES /2/4/···/32
TRNG_CLK 1M
ADC1MSEL
ADC1MPRES /1/2/···/32
ADC1/2/3/4_CLK 1M
FLASH_CLK to Flash programming
RNGCPRES /1/2/3/4/···/32
RNGC_CLK
USBHS_240M
PLL
ADCOTGPLLSEL
ADCOTGPLLPRES /1/2/···/12
ADC1/2/3/4_CLK
ADCHPRES /1/2/···/32
CKMOD
TRACEPRES /1/2/4/8
TRACE_CLK
FCLK
HCLK
CPU AHB BUS
/8
SysTick
HCLKPRES /1/2
HCLK to SAC/SDIO/USBHS/ETH
APB1 Prescaler /1/2/4/8/16
180MHz MAX
PCLK1 to APB1 peripherals
If(APB1 Prescaler = 1) x1 else x2
GTIM1/2/3/4/5/6/7_CLK BTIM1/2_CLK LPTIM1/2_CLK
APB2 Prescaler /1/2/4/8/16
180MHz MAX
PCLK2 to APB2 peripherals
If(APB2 Prescaler = 1) x1 else x2
SYSCLK
GTIM8/9/10_CLK
GTIMCLKSEL
HCLK
SYSCLK
ATIM1/2/3_CLK
ATIMCLKSEL
AHB Prescaler /1/2/···/512
HCLK 240MHz MAX
SYSCLK 240MHz MAX
SCLKSW
HSI
HSE
PLL
SHRTPLL
SCLK
PLLSEL
PLLHSI PRES /1/2
PLLHSE PRES /1/2
PLLSRC
PLLINP RES /1/4
PLLMULFCT x4...x76
PLLOUT PRES /1...8
HSI RC 8MHz
HSE OSC 4~32MHz
OSC_OUT
OSC_IN
LSE OSC 32.768KHz
OSC32_OUT
OSC32_IN
LSI RC 32KHz
/128
LSE
LSI
RTCSEL
RTC_CLK
IWDG_CLK
HSI
HSE
LPTIM_CLK (PCLK1)
LPTIM1/2SEL
LPTIM1/2_CLK
I2S2/3_CLKIN
SYSCLK
SHRTPLL
HSI
I2S2/3SEL
I2S2/3_CLK
ETH_MII_TX_CLK
ETH_RMII_REF_CLK
ETH_MII_RX_CLK
/20
/2
MII_RMII_SEL
MAC_SPEED
eth_mii_tx_clk
eth_mii_rx_clk
eth_rmii_clk
USBHSSEL
HSE
USBHS PLL
USBHS_240M
USBHS_CLK(60M)
USBOHS_48M
UCDR
USBFS_CLK(48M)
USBFSSEL
USBPLLSEL
SHRTPLL
PLL
USBPLLPRES /1/2/3.../15
USBPLL_DIVCLK
SHRTPLL
PLL
FDCANPLLPRES /1/2/3.../15
FDCANPLLPRES /1/2/3.../15
FDCAN_CLK
HSE
FDCANSEL
MCO2
SYSCLK
HSI
HSE
LSI
LSE
MCO2
PLL_MCOPRES /2/3/4/.../15
PLL
SHRTPLL
USBHS_240M
MCOPLLSEL
MCO1
SYSCLK
HSI
HSE
LSI
LSE
MCO1

4.3.2 PWR

下表是N32G45x系列和N32H47x_48x系列在PWR模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用PWR模块驱动API函数可以依照下表做替换：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
PWR 复位	void PWR_DeInit(void)	void PWR_DeInit(void)	是	无
RTC 和备份寄存器访问使能	void PWR_BackupAccessEnable(FunctionalState Cmd)	void PWR_BackupAccessEnable(FunctionalState Cmd)	是	无
PVD 使能	void PWR_PvdEnable(FunctionalState Cmd)	void PWR_PvdEnable(FunctionalState Cmd)	是	无
PVD 阈值档位配置	void PWR_PvdRangeConfig(uint32_t PWR_PVDLevel)	void PWR_PVDLevelConfig(uint32_t level)	否	函数命名及输入参数 PWR_PVDLevel 不一致： N32G45x 系列参数： PWR_PVDRANGRE_2V2 PWR_PVDRANGRE_2V3 PWR_PVDRANGRE_2V4 PWR_PVDRANGRE_2V5 PWR_PVDRANGRE_2V6 PWR_PVDRANGRE_2V7 PWR_PVDRANGRE_2V8 PWR_PVDRANGRE_2V9 N32H47x_48x 系列参数： PWR_PVD_LEVEL_2V18 PWR_PVD_LEVEL_2V28 PWR_PVD_LEVEL_2V38 PWR_PVD_LEVEL_2V48 PWR_PVD_LEVEL_2V58 PWR_PVD_LEVEL_2V68 PWR_PVD_LEVEL_2V78 PWR_PVD_LEVEL_2V88 PWR_PVD_LEVEL_1V78 PWR_PVD_LEVEL_1V88 PWR_PVD_LEVEL_1V98 PWR_PVD_LEVEL_2V08 PWR_PVD_LEVEL_3V28 PWR_PVD_LEVEL_3V38 PWR_PVD_LEVEL_3V48 PWR_PVD_LEVEL_3V58

PVD 检测源配置	-	void PWR_PVDSourceConfig(uint32_t PVD_source)	否	N32G45x 系列无此 API
唤醒引脚使能	void PWR_WakeUpPinEnable(FunctionalState Cmd)	void PWR_WakeUpPinEnable(uint32_t pin, FunctionalState Cmd)	否	N32H47x_48x 系列新增输入参数 pin： WAKEUP_PIN0EN WAKEUP_PIN1EN WAKEUP_PIN2EN WAKEUP_PIN3EN WAKEUP_PIN4EN WAKEUP_PIN5EN
唤醒引脚极性配置	-	void PWR_WakeUpPinPolarity(uint32_t pin, WAKEUP_PIN_POL polarity)	否	N32G45x 系列无此 API
唤醒外设使能	-	void PWR_WakeUpPeriphEnable(uint32_t periph, FunctionalState Cmd)	否	N32G45x 系列无此 API
进入 STOP 模式	void PWR_EnterStopState(uint32_t PWR_Regulator, uint8_t PWR_STOPEntry)	void PWR_EnterSTOPMode(uint32_t PWR_Regulator, uint8_t PWR_STOPEntry)	否	函数命名及输入参数 PWR_Regulator 不一致： PWR_REGULATOR_ON→PWR_REGULATOR_NORMAL
进入 SLEEP 模式	void PWR_EnterSLEEPMode(uint8_t SLEEPONEXIT, uint8_t PWR_STOPEntry)	void PWR_EnterSLEEPMode(PWR_SLEEPONEXIT_STATUS SLEEPONEXIT, uint8_t PWR_SLEEPPEntry)	否	输入参数不一致： 参数 SLEEPONEXIT： 0→PWR_SLEEP_NOW 1→PWR_SLEEP_ON_EXIT 参数 PWR_SLEEPPEntry： PWR_STOPENTRY_WFI→PWR_SLEEPENTRY_WFI PWR_STOPENTRY_WFE→PWR_SLEEPENTRY_WFE
进入 STOP2 模式	void PWR_EnterSTOP2Mode(uint8_t PWR_STOPEntry)	-	否	N32H47x_48x 系列无此 API
进入 STANDBY 模式	void PWR_EnterStandbyState(void)	void PWR_EnterSTANDBYMode(void)	否	函数命名不一致

PWR 状态读取	FlagStatus PWR_GetFlagStatus(uint32_t PWR_FLAG)	FlagStatus PWR_GetFlagStatus(uint32_t PWR_FLAG)	否	输入参数 PWR_FLAG 不一致： N32G45x 系列参数： PWR_WU_FLAG PWR_SB_FLAG PWR_PVDO_FLAG PWR_VBATF_FLAG N32H47x_48x 系列参数： PWR_FLAG_WKUP0 PWR_FLAG_WKUP1 PWR_FLAG_WKUP2 PWR_FLAG_WKUP3 PWR_FLAG_WKUP4 PWR_FLAG_WKUP5 PWR_FLAG_WKUPP PWR_FLAG_STANDBY PWR_FLAG_PVDOUT PWR_FLAG_VBAT
PWR 状态清除	void PWR_ClearFlag(uint32_t PWR_FLAG)	void PWR_ClearFlag(uint32_t PWR_FLAG)	否	输入参数 PWR_FLAG 不一致： N32G45x 系列参数： PWR_WU_FLAG PWR_SB_FLAG N32H47x_48x 系列参数： PWR_CLR_WKUP0 PWR_CLR_WKUP1 PWR_CLR_WKUP2 PWR_CLR_WKUP3 PWR_CLR_WKUP4 PWR_CLR_WKUP5 PWR_CLR_WKUPP PWR_CLR_STANDBY PWR_CLR_VBAT
BKR 电压调节	-	void PWR_ConfigBKROutput(uint32_t voltage_output)	否	N32G45x 系列无此 API
IWDG 复位使能	-	void PWR_EnableIWDGReset(FunctionalState Cmd)	否	N32G45x 系列无此 API
Standby 模式 BRPSRAM 保持配置	-	void PWR_EnableBKPSRAMRetainInStandbyMode(FunctionalState Cmd)	否	N32G45x 系列无此 API
Vbat 模式 BRPSRAM 保持配置	-	void PWR_EnableBKPSRAMRetainInVbatMode(FunctionalState Cmd)	否	N32G45x 系列无此 API

LSE 噪声优化使能	-	void PWR_EnableLSENoiseMitigation(FunctionalState Cmd)	否	N32G45x 系列无此 API
晶体模式 LSE 中断使能	-	void PWR_EnableLSECSSCrystalInt(FunctionalState Cmd)	否	N32G45x 系列无此 API
旁路模式 LSE CSS 高限制值配置	-	void PWR_ConfigLSECSSBypassHighLimit(uint32_t high_limit)	否	N32G45x 系列无此 API
LSE 失效时 RTC 源切换配置	-	void PWR_EnableLSECSSRTCsourceSwitch(FunctionalState Cmd)	否	N32G45x 系列无此 API
LSE CSS 状态获取	-	FlagStatus PWR_GetLSECSSFlagStatus(uint32_t PWR_FLAG)	否	N32G45x 系列无此 API
LSE CSS 状态清除	-	void PWR_ClearLSECSSFlag(void)	否	N32G45x 系列无此 API
LCO 时钟源配置	-	void PWR_ConfigLco(uint32_t LCO_source)	否	N32G45x 系列无此 API
LCO 使能	-	void PWR_EnableLCO(FunctionalState Cmd)	否	N32G45x 系列无此 API
NRST 引脚配置	-	void PWR_ConfigNRSTMode(uint32_t NRST_mode)	否	N32G45x 系列无此 API

下图是N32G45x系列（左）和N32H47x_48x系列（右）在PWR模块的“SLEEP”示例工程代码对比：

```c
 * @brief  Main program.
 */
int main(void)
{
    /*!< At this stage the microcontroller clock setting is already configured,
         this is done through SystemInit() function which is called from startup
         file (startup_n32g45x_xx.s) before to branch to application main.
         To reconfigure the default setting of SystemInit() function, refer to
         system_n32g45x.c file
    */
    log_init();
    log_info("\r\n MCU Reset! \r\n");
    /* Enable PWR Clock */
    RCC_EnableAPB1PeriphClk(RCC_APB1_PERIPH_PWR, ENABLE);
    /* Initialize LEDs on n32g45x-EVAL board */
    LEDInit(LED1_PORT,LED1_PIN);
    LEDInit(LED3_PORT,LED3_PIN);
    LEDOff(LED1_PORT,LED1_PIN);
    LEDOff(LED3_PORT,LED3_PIN);
    /* Initialize Key button Interrupt to wake up the low power*/
    KeyInputExtiInit(KEY_INPUT_PORT, KEY_INPUT_PIN);
    /* Clear the Interrupt flag */
    EXTI_ClrITPendBit(EXTI_LINE0);
    DBG_ConfigPeriph(DBG_SLEEP, ENABLE);
    while (1)
    {
       /* Turn on LED3 */
       LEDBlink(LED3_PORT,LED3_PIN);
       /* Insert a long delay */
       delay(60);
       /* Request to enter SLEEP mode*/
       PWR_EnterSLEEPMode(SLEEP_NOW, PWR_STOPENTRY_WFI);
    }
}
```

```c
int main(void)
{
    /*   At this stage the microcontroller clock setting is already configured,
         this is done through SystemInit() function which is called from startup
         file (startup_n32h4xx.s) before to branch to application main.
         To reconfigure the default setting of SystemInit() function, refer to
         system_n32h47x_48x.c file
    */
    log_init();
    log_info("\r\n --- Reset --- \r\n");

    /* Enable PWR Clock */
    RCC_EnableAPB1PeriphClk(RCC_APB1_PERIPH_PWR, ENABLE);
    /* Initialize Key button Interrupt to wake up the low power*/
    KeyInputExtiInit(KEY_INPUT_PORT, KEY_INPUT_PIN);
    /* Clear the Interrupt flag */
    EXTI_ClrITPendBit(KEY_EXTI_LINE);
    /*  Enable the DBG_SLEEP to keep debug in low power  */
    //DBG_ConfigPeriph(DBG_SLEEP, ENABLE);
    while (1)
    {
       /* Insert a long delay */
       delay(700);
       log_info("Entry SLEEP mode\n");
       /* Request to enter SLEEP mode*/
       PWR_EnterSLEEPMode(PWR_SLEEP_NOW, PWR_SLEEPENTRY_WFI);
       log_info("Exit SLEEP mode\n");
    }
}

/**
 *\*\name    delay.
 *\*\fun     Delay function.
```

PWR配置流程基本一致，区别在于N32H47x_48x系列将LED唤醒提示修改为打印提示，并且屏蔽了DBG_SLEEP模式配置，用户可根据是否需要进调式模式打开。

4.3.3 DMA

下表是N32G45x系列和N32H47x_48x系列在DMA模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用DMA模块驱动API函数可以依照下表做替换：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
DMA 复位	void DMA_DeInit(DMA_ChannelType* DMAyChx)	void DMA_DeInit(DMA_ChannelType* DMAChx)	是	无
DMA 初始化	void DMA_Init(DMA_ChannelType* DMAyChx, DMA_InitType* DMA_InitParam)	void DMA_Init(DMA_ChannelType* DMAChx, DMA_InitType* DMA_InitParam)	否	新增 burst 功能，初始化前用户应按需配置 DMA_InitParam 参数中的 BURST_BYPASS、BURST_MODE 以及 BURST_LEN
DMA 配置结构体参数初始化	void DMA_StructInit(DMA_InitType* DMA_InitParam)	void DMA_StructInit(DMA_InitType* DMA_InitParam)	是	无
DMA 通道使能	void DMA_EnableChannel(DMA_ChannelType* DMAyChx, FunctionalState Cmd)	void DMA_EnableChannel(DMA_ChannelType* DMAChx, FunctionalState Cmd)	是	无
DMA 中断配置	void DMA_ConfigInt(DMA_ChannelType* DMAyChx, uint32_t DMAInt, FunctionalState Cmd)	void DMA_ConfigInt(DMA_ChannelType* DMAChx, uint32_t DMAInt, FunctionalState Cmd)	是	无
设置当前传输计数（要传输的数量）	void DMA_SetCurrDataCounter(DMA_ChannelType* DMAyChx, uint16_t DataNumber)	void DMA_SetCurrDataCounter(DMA_ChannelType* DMAChx, uint16_t DataNumber)	是	无
获取当前传输计数（剩余的待传输数量）	uint16_t DMA_GetCurrDataCounter(DMA_ChannelType* DMAyChx)	uint16_t DMA_GetCurrDataCounter(DMA_ChannelType* DMAChx)	是	无
获取标志位状态	FlagStatus DMA_GetFlagStatus(uint32_t DMAyFlag, DMA_Module* DMAy)	FlagStatus DMA_GetFlagStatus(uint32_t DMAFlag, DMA_Module *DMAy)	否	输入的第一个实参不一致，N32H47x_48x 系列输入如下参数之一： DMA_FLAG_GL1 DMA_FLAG_TC1 DMA_FLAG_HT1 DMA_FLAG_TE1 DMA_FLAG_GL2 DMA_FLAG_TC2 DMA_FLAG_HT2 DMA_FLAG_TE2 DMA_FLAG_GL3 DMA_FLAG_TC3 DMA_FLAG_HT3

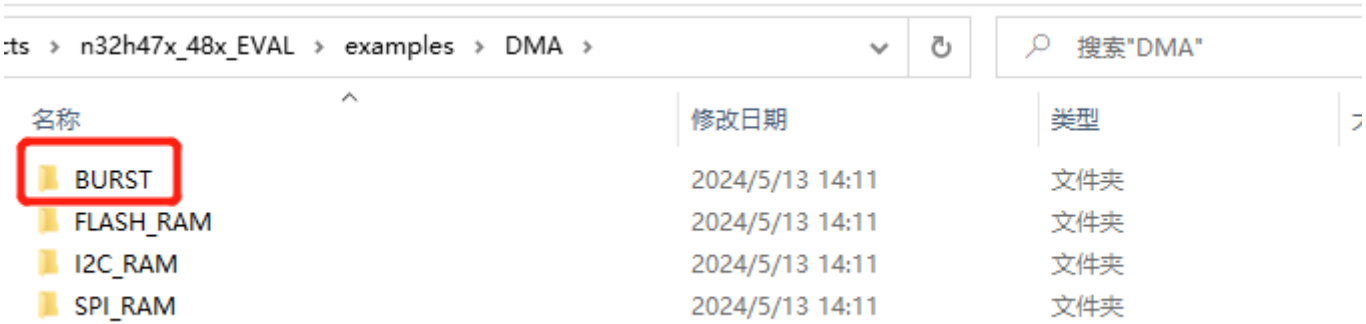
API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
				DMA_FLAG_TE3 DMA_FLAG_GL4 DMA_FLAG_TC4 DMA_FLAG_HT4 DMA_FLAG_TE4 DMA_FLAG_GL5 DMA_FLAG_TC5 DMA_FLAG_HT5 DMA_FLAG_TE5 DMA_FLAG_GL6 DMA_FLAG_TC6 DMA_FLAG_HT6 DMA_FLAG_TE6 DMA_FLAG_GL7 DMA_FLAG_TC7 DMA_FLAG_HT7 DMA_FLAG_TE7 DMA_FLAG_GL8 DMA_FLAG_TC8 DMA_FLAG_HT8 DMA_FLAG_TE8
清除标志位	void DMA_ClearFlag(uint32_t DMAFlag, DMA_Module *DMAy)	void DMA_ClearFlag(uint32_t DMAyFlag, DMA_Module* DMAy)	否	输入的第一个实参不一致，N32H47x_48x 系列输入如下参数之一： DMA_FLAG_GL1 DMA_FLAG_TC1 DMA_FLAG_HT1 DMA_FLAG_TE1 DMA_FLAG_GL2 DMA_FLAG_TC2 DMA_FLAG_HT2 DMA_FLAG_TE2 DMA_FLAG_GL3 DMA_FLAG_TC3 DMA_FLAG_HT3 DMA_FLAG_TE3 DMA_FLAG_GL4 DMA_FLAG_TC4 DMA_FLAG_HT4 DMA_FLAG_TE4 DMA_FLAG_GL5

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
				DMA_FLAG_TC5 DMA_FLAG_HT5 DMA_FLAG_TE5 DMA_FLAG_GL6 DMA_FLAG_TC6 DMA_FLAG_HT6 DMA_FLAG_TE6 DMA_FLAG_GL7 DMA_FLAG_TC7 DMA_FLAG_HT7 DMA_FLAG_TE7 DMA_FLAG_GL8 DMA_FLAG_TC8 DMA_FLAG_HT8 DMA_FLAG_TE8
获取中断状态	INTStatus DMA_GetIntStatus(uint32_t DMAy_Int, DMA_Module *DMAy)	INTStatus DMA_GetIntStatus(uint32_t DMAy_IT, DMA_Module* DMAy)	否	输入的第一个实参不一致，N32H47x_48x 系列输入如下参数之一： DMA_INT_GLB1 DMA_INT_TXC1 DMA_INT_HTX1 DMA_INT_ERR1 DMA_INT_GLB2 DMA_INT_TXC2 DMA_INT_HTX2 DMA_INT_ERR2 DMA_INT_GLB3 DMA_INT_TXC3 DMA_INT_HTX3 DMA_INT_ERR3 DMA_INT_GLB4 DMA_INT_TXC4 DMA_INT_HTX4 DMA_INT_ERR4 DMA_INT_GLB5 DMA_INT_TXC5 DMA_INT_HTX5 DMA_INT_ERR5 DMA_INT_GLB6 DMA_INT_TXC6 DMA_INT_HTX6

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
				DMA_INT_ERR6 DMA_INT_GLB7 DMA_INT_TXC7 DMA_INT_HTX7 DMA_INT_ERR7 DMA_INT_GLB8 DMA_INT_TXC8 DMA_INT_HTX8 DMA_INT_ERR8
清除中断挂起位	void DMA_ClrIntPendingBit(uint32_t DMAy_IT, DMA_Module* DMAy)	void DMA_ClrIntPendingBit(uint32_t DMAy_Int, DMA_Module *DMAy)	否	输入的第一个实参不一致，N32H47x_48x 系列输入如下参数之一： DMA_INT_GLB1 DMA_INT_TXC1 DMA_INT_HTX1 DMA_INT_ERR1 DMA_INT_GLB2 DMA_INT_TXC2 DMA_INT_HTX2 DMA_INT_ERR2 DMA_INT_GLB3 DMA_INT_TXC3 DMA_INT_HTX3 DMA_INT_ERR3 DMA_INT_GLB4 DMA_INT_TXC4 DMA_INT_HTX4 DMA_INT_ERR4 DMA_INT_GLB5 DMA_INT_TXC5 DMA_INT_HTX5 DMA_INT_ERR5 DMA_INT_GLB6 DMA_INT_TXC6 DMA_INT_HTX6 DMA_INT_ERR6 DMA_INT_GLB7 DMA_INT_TXC7 DMA_INT_HTX7 DMA_INT_ERR7 DMA_INT_GLB8

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
				DMA_INT_TXC8 DMA_INT_HTX8 DMA_INT_ERR8
设置通道请求映射	void DMA_RequestRemap(uint32_t DMAy_REMAP, DMA_Module* DMAy, DMA_ChannelType* DMAyChx, FunctionalState Cmd)	void DMA_RequestRemap(uint32_t DMA_Remap, DMA_ChannelType* DMAChx, FunctionalState Cmd)	否	N32H47x_48x 系列无 DMAy 参数； 输入的第一个实参不一致，使用时，用户根据所用的外设按需输入参数即可，映射关系多达 148 种，此处不一一罗列，具体请参考用户手册的通道选择寄存器或 DMA 驱动头文件的相关宏定义。

N32H47x_48x系列新增burst功能，SDK中也同步新增对应的示例工程，如需使用该功能，用户可参考SDK中的BURST示例，如下图所示：



如不使用该功能，用户必须遵循用户手册中“通道配置流程”章节的相关描述。

4.3.4 CRC

下表是N32G45x系列和N32H47x_48x系列在CRC模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用CRC模块驱动API函数可以依照下表做替换：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
CRC 复位	void CRC32_ResetCrc(void);	void CRC32_ResetCrc(void);	是	无
计算给定数据字(32 位)的 32 位 CRC	uint32_t CRC32_CalcCrc(uint32_t Data);	uint32_t CRC32_CalcCrc(uint32_t Data);	是	无
计算给定数据字缓冲区的 32 位 CRC(32 位)	uint32_t CRC32_CalcBufCrc(uint32_t pBuffer[], uint32_t BufferLength);	uint32_t CRC32_CalcBufCrc(const uint32_t pBuffer[], uint32_t BufferLength);	是	无
获取当前的 32bit CRC 值	uint32_t CRC32_GetCrc(void);	uint32_t CRC32_GetCrc(void);	是	无
在独立数据(ID)寄存器中存储 8 位数据	Void CRC32_SetIDat(uint8_t IDValue);	void CRC32_SetIDat(uint8_t IDValue);	是	无
获取独立数据(ID)寄存器中存储 8 位数据	uint8_t CRC32_GetIDat(void);	uint8_t CRC32_GetIDat(void);	是	无
计算给定 byte 数据 (8 位)的 16 位 CRC	uint16_t CRC16_CalcCRC(uint8_t Data);	void PWR_WakeUpPinPolarity(uint32_t pin, WAKEUP_PIN_POL polarity)	是	无
计算给定 byte 数据缓冲区的 16 位 CRC(16 位)	uint16_t CRC16_CalcBufCrc(uint8_t pBuffer[], uint32_t BufferLength);	uint16_t CRC16_CalcBufCrc(const uint8_t pBuffer[], uint32_t BufferLength);	是	无
在 16 位 CRC 数据寄存器中写入 8 位数据。	-	void __CRC16_SetCrc(uint8_t Data);	否	N32G45x 系列无此 API
获取当前的 16bit CRC 值	-	uint16_t __CRC16_GetCrc(void);	否	N32G45x 系列无此 API
写校验寄存器初始值(LRC)	-	void __CRC16_SetLRC(uint8_t Data);	否	N32G45x 系列无此 API
获取当前校验寄存器值(LRC)	-	uint8_t __CRC16_GetLRC(void)	否	N32G45x 系列无此 API
将计算数据设置为小端格式	-	void __CRC16_SetLittleEndianFmt(void);	否	N32G45x 系列无此 API
将计算数据设置为大端格式	-	void __CRC16_SetBigEndianFmt(void);	否	N32G45x 系列无此 API
Enable Clean CRC16 value	-	void __CRC16_SetCleanEnable(void);	否	N32G45x 系列无此 API
Disable Clean CRC16 value	-	void __CRC16_SetCleanDisable(void);	否	N32G45x 系列无此 API

CRC使用流程基本一致， N32H47x_48x系列提供了16bit、32bit CRC的计算示例，还提供了通过DMA传输数据计算CRC值得示例，用户可以参考示例代码。

```
183 int main(void)
184 {
185     /* Enable CRC clock */
186     RCC_EnableAHBPeriphClk(RCC_AHB_PERIPHEN_CRC, ENABLE);
187 
188     log_init();
189 
190     /* Compute the 32bit CRC of "DataBuffer" */
191     CRC32Value = CRC32_CalcBufCrc((uint32_t*)DataBuffer, BUFFER_SIZE);
192 
193     /* Compute the 32bit CRC of "DataBuffer" by software*/
194     CRCValue   = GetCRC32_Software((uint32_t*)DataBuffer, BUFFER_SIZE ,0xffffffff);
195 
196     if( CRC32Value == CRCValue )
197     {
198         printf(" CRC32 calculation is correct! \r\n");
199     }else
200     {
201         printf(" CRC32 calculation error! \r\n");
202     }
203 
204     /* Compute the 16bit CRC of an array */
205     CRC16Value = CRC16_CalcBufCrc((uint8_t*)Buffer, 8);
206 
207     /* Compute the 16bit CRC of an array by software*/
208     CRCValue   = GetCRC16_Software((uint8_t*)Buffer, 8, 0x00);
209 
210     if( CRC16Value == CRCValue )
211     {
212         printf("CRC16 calculation is correct! \r\n");
213     }else
214     {
215         printf("CRC16 calculation error! \r\n");
216     }
217 
218     while (1)
219     {
220     }
221 }
222 
```

```
69  int main(void)
70  {
71      /* Enable CRC and DMA clock */
72      RCC_EnableAHBPeriphClk(RCC_AHB_PERIPHEN_CRC|RCC_AHB_PERIPHEN_DMA1, ENABLE);
73  
74      log_init();
75  
76      printf(" CRC DMA test start.\r\n");
77  
78      CRC32_ResetCrc();
79  
80      TestCrc32AndDma(0,2);
81      TestCrc16AndDma(0,2);
82  
83      while (1)
84      {
85      }
86  }
87  
```

4.3.5 SPI/I2S

下表是N32G45x系列和N32H47x_48x系列在SPI/I2S模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用SPI/I2S模块驱动API函数可以依照下表做替换：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
SPI 复位	void SPI_I2S_DeInit(SPI_Module* SPIx);	void SPI_I2S_DeInit(const SPI_Module* SPIx);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
SPI 接口初始化	void SPI_Init(SPI_Module* SPIx, SPI_InitType* SPI_InitStruct);	void SPI_Init(SPI_Module* SPIx,const SPI_InitType* SPI_InitStruct);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
I2S 接口初始化	void I2S_Init(SPI_Module* SPIx, I2S_InitType* I2S_InitStruct);	void I2S_Init(SPI_Module* SPIx,const I2S_InitType* I2S_InitStruct);	是	无
初始化 SPI_InitStruct 成员。	void SPI_InitStruct(SPI_InitType* SPI_InitStruct);	void SPI_InitStruct(SPI_InitType* SPI_InitStruct);	是	无
初始化 I2S_InitStruct 成员。	void I2S_InitStruct(I2S_InitType* I2S_InitStruct);	void I2S_InitStruct(I2S_InitType* I2S_InitStruct);	是	无
使能 SPI	void SPI_Enable(SPI_Module* SPIx, FunctionalState Cmd);	void SPI_Enable(SPI_Module* SPIx, FunctionalState Cmd);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
使能 I2S	void I2S_Enable(SPI_Module* SPIx, FunctionalState Cmd);	void I2S_Enable(SPI_Module* SPIx, FunctionalState Cmd);	是	无
启用或禁用指定的 SPI/I2S 中断。	void SPI_I2S_EnableInt(SPI_Module* SPIx, uint8_t SPI_I2S_IT, FunctionalState Cmd);	void SPI_I2S_EnableInt(SPI_Module* SPIx, uint8_t SPI_I2S_IT, FunctionalState Cmd);	否	输入参数 SPIx，SPI_I2S_IT 不一致： N32G45x 系列 SPIx 参数： SPI1/SPI2/SPI3 N32H47x_48x 系列 SPIx 参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6 N32G45x 系列 SPI_I2S_IT 参数： SPI_I2S_INT_TE

				SPI_I2S_INT_RNE SPI_I2S_INT_ERR N32H47x_48x 系列 SPI_I2S_IT 参数： SPI_I2S_INT_TE SPI_I2S_INT_RNE SPI_I2S_INT_ERR SPI_I2S_INT_RXONLYC SPI_I2S_INT_RXFIFOF SPI_I2S_INT_RXFIFOHF SPI_I2S_INT_TXFIFOHE
启用或禁用 SPIx/I2Sx DMA 接口	void SPI_I2S_EnableDma(SPI_Module* SPIx, uint16_t SPI_I2S_DMAReq, FunctionalState Cmd);	void SPI_I2S_EnableDma(SPI_Module* SPIx, uint16_t SPI_I2S_DMAReq, FunctionalState Cmd);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
通过 SPIx/I2Sx 传输数据	void SPI_I2S_TransmitData(SPI_Module* SPIx, uint16_t Data);	void SPI_I2S_TransmitData(SPI_Module* SPIx, uint16_t Data);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
通过 SPIx/I2Sx 接收数据	uint16_t SPI_I2S_ReceiveData(SPI_Module* SPIx);	void SPI_SetNssLevel(SPI_Module* SPIx, uint16_t SPI_NSSInternalSoft);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
通过软件在内部配置所选 SPI 的 NSS 引脚	void SPI_SetNssLevel(SPI_Module* SPIx, uint16_t SPI_NSSInternalSoft);	void SPI_SetNssLevel(SPI_Module* SPIx, uint16_t SPI_NSSInternalSoft);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
启用或禁用所选 SPI 的 SS 输出	void SPI_SSOutputEnable(SPI_Module* SPIx, FunctionalState Cmd);	void SPI_SSOutputEnable(SPI_Module* SPIx, FunctionalState Cmd);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
配置所选 SPI 的数据长度	void SPI_ConfigDataLen(SPI_Module* SPIx, uint16_t DataLen);	void SPI_ConfigDataLen(SPI_Module* SPIx, uint16_t DataLen);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6

传输 SPIx CRC 值。	void SPI_TransmitCrcNext(SPI_Module* SPIx);	void SPI_TransmitCrcNext(SPI_Module* SPIx, FunctionalState Cmd);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
启用或禁用传输字节的 CRC 值计算。	void SPI_EnableCalculateCrc(SPI_Module* SPIx, FunctionalState Cmd);	void SPI_EnableCalculateCrc(SPI_Module* SPIx, FunctionalState Cmd);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
获取指定 SPI 的发送或接收 CRC 寄存器值	uint16_t SPI_GetCRCDat(SPI_Module* SPIx, uint8_t SPI_CRC);	uint16_t SPI_GetCRCDat(const SPI_Module* SPIx, uint8_t SPI_CRC);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
获取指定 SPI 的 CRC 多项式寄存器值	uint16_t SPI_GetCRCPoly(SPI_Module* SPIx);	uint16_t SPI_GetCRCPoly(const SPI_Module* SPIx);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
选择双向模式下的数据传输方向。	void SPI_ConfigBidirectionalMode(SPI_Module* SPIx, uint16_t DataDirection);	void SPI_ConfigBidirectionalMode(SPI_Module* SPIx, uint16_t DataDirection);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
检查指定的 SPI/I2S 标志是否已设置。	FlagStatus SPI_I2S_GetStatus(SPI_Module* SPIx, uint16_t SPI_I2S_FLAG);	FlagStatus SPI_I2S_GetStatus(const SPI_Module* SPIx, uint16_t SPI_I2S_FLAG);	否	输入参数 SPIx，SPI_I2S_FLAG 不一致： N32G45x 系列 SPIx 参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6 N32G45x 系列 SPI_I2S_FLAG 参数： SPI_I2S_TE_FLAG SPI_I2S_RNE_FLAG SPI_I2S_BUSY_FLAG SPI_I2S_OVER_FLAG SPI_MODERR_FLAG SPI_CRCERR_FLAG I2S_UNDER_FLAG

				I2S_CHSIDE_FLAG N32H47x_48x 系列 SPI_I2S_FLAG 参数： SPI_I2S_BUSY_FLAG SPI_I2S_OVER_FLAG SPI_MODERR_FLAG SPI_CRCERR_FLAG I2S_UNDER_FLAG I2S_CHSIDE_FLAG SPI_I2S_TE_FLAG SPI_I2S_RNE_FLAG SPI_I2S_RXONLYC_FLAG SPI_I2S_RXFIFOF_FLAG SPI_I2S_TXFIFOE_FLAG SPI_I2S_RXFIFOHF_FLAG SPI_I2S_TXFIFOHE_FLAG
清除 SPIx CRC 错误（CRCERR）标志	void SPI_I2S_ClrCRCErrFlag(SPI_Module* SPIx, uint16_t SPI_I2S_FLAG);	void SPI_I2S_ClrCRCErrFlag(SPI_Module* SPIx, uint16_t SPI_I2S_FLAG);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
检查指定的 SPI/I2S 中断是否已发生	INTStatus SPI_I2S_GetIntStatus(SPI_Module* SPIx, uint8_t SPI_I2S_IT);	INTStatus SPI_I2S_GetIntStatus(const SPI_Module* SPIx, uint8_t SPI_I2S_IT);	否	输入参数 SPIx，SPI_I2S_IT 不一致： N32G45x 系列 SPIx 参数： SPI1/SPI2/SPI3 N32H47x_48x 系列 SPIx 参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6 N32G45x 系列 SPI_I2S_IT 参数： SPI_I2S_INT_TE SPI_I2S_INT_RNE SPI_I2S_INT_OVER SPI_INT_MODERR SPI_INT_CRCERR I2S_INT_UNDER N32H47x_48x 系列 SPI_I2S_IT 参数： SPI_I2S_INT_TE SPI_I2S_INT_RNE SPI_I2S_INT_ERR SPI_I2S_INT_RXONLYC SPI_I2S_INT_RXFIFOF

				SPI_I2S_INT_RXFIFOHF SPI_I2S_INT_TXFIFOHE
清除 SPIx CRC 错误（CRCERR）中断挂起位	void SPI_I2S_ClrITPendingBit(SPI_Module* SPIx, uint8_t SPI_I2S_IT);	void SPI_I2S_ClrITPendingBit(SPI_Module* SPIx, uint8_t SPI_I2S_IT);	否	输入参数 SPIx 不一致： N32G45x 系列参数： SPI1/SPI2/SPI3 N32H47x_48x 系列参数： SPI1/SPI2/SPI3/SPI4/SPI5/SPI6
启用或禁用 SPI FIFO	-	void SPI_I2S_FIFO_Cmd(SPI_Module* SPIx, FunctionalState NewState);	否	N32G45x 系列无该接口
清除 SPIx FIFO 位	-	void SPI_I2S_ClearFIFOBit(SPI_Module* SPIx, uint16_t SPI_I2S_FIFO_Clear);	否	N32G45x 系列无该接口
配置 SPI Rx FIFO 大小	-	void SPI_RxFIFOSizeConfig(SPI_Module* SPIx, uint16_t SPI_FIFOSize)	否	N32G45x 系列无该接口
配置 SPI Tx FIFO 大小	-	void SPI_TxFIFOSizeConfig(SPI_Module* SPIx, uint16_t SPI_FIFOSize)	否	N32G45x 系列无该接口
获取当前有效接收 FIFO 的数量	-	uint16_t SPI_RX_FIFO_CNT_GET(const SPI_Module* SPIx);	否	N32G45x 系列无该接口
获取当前有效发送 FIFO 的数量	-	uint16_t SPI_TX_FIFO_CNT_GET(const SPI_Module* SPIx);	否	N32G45x 系列无该接口
设置要传输的数据数量		void SPI_TRANSNUM_SET(SPI_Module* SPIx, uint16_t Data);	否	N32G45x 系列无该接口
获取要传输的数据数量	-	uint16_t SPI_TRANSNUM_GET(const SPI_Module* SPIx);	否	N32G45x 系列无该接口
SPI 主时钟延迟时间配置	-	void SPI_DELAYTIME_SET(SPI_Module* SPIx, uint16_t Data);	否	N32G45x 系列无该接口
获取 SPI 主时钟延迟时间	-	uint16_t SPI_DELAYTIME_GET(const SPI_Module* SPIx);	否	N32G45x 系列无该接口
将要发送的数据写入 RX FIFO	-	void SPI_RX_FIFO_SET(SPI_Module* SPIx, uint16_t Data);	否	N32G45x 系列无该接口
FIFO 模式下，接收数据	-	uint16_t SPI_RX_FIFO_GET(const SPI_Module* SPIx);	否	N32G45x 系列无该接口
根据 I2S_EXT_InitStruct 中的指定参数初始化 I2Sx_EXT 外围设备。	-	void I2S_EXT_Init(I2S_EXT_Module* I2Sx_EXT,const I2S_InitType* I2S_EXT_InitStruct);	否	N32G45x 系列无该接口
启用或禁用指定的 I2S EXT 外围设备	-	void I2S_EXT_Enable(I2S_EXT_Module* I2Sx_EXT, FunctionalState Cmd);	否	N32G45x 系列无该接口
通过 I2Sx_EXT 外围设备传输数据	-	void I2S_EXT_TransmitData(I2S_EXT_Module* I2Sx, uint16_t Data);	否	N32G45x 系列无该接口
通过 I2Sx_EXT 外围设备接收数据	-	uint16_t I2S_EXT_ReceiveData(const I2S_EXT_Module* I2Sx);	否	N32G45x 系列无该接口
使能或者禁止 I2Sx DMA 接口	-	void I2S_EXT_EnableDma(I2S_EXT_Module* I2Sx, uint16_t I2S_EXT_DMAReq, FunctionalState Cmd);	否	N32G45x 系列无该接口

使能或者禁止 I2Sx 中断	-	void I2S_EXT_EnableInt(I2S_EXT_Module* I2Sx, uint8_t I2S_EXT_IT, FunctionalState Cmd);	否	N32G45x 系列无该接口
获取指定的 I2S_EXT 标志是否置起	-	FlagStatus I2S_EXT_GetStatus(const I2S_EXT_Module* I2Sx, uint16_t I2S_EXT_FLAG);	否	N32G45x 系列无该接口
获取指定的 I2S_EXT 中断标志是否置起	-	INTStatus I2S_EXT_GetIntStatus(const I2S_EXT_Module* I2Sx, uint8_t I2S_EXT_IT);	否	N32G45x 系列无该接口

N32H47x_48x系列和N32G45x系列SPI使用流程基本一致，但是他们的宏定义差别很大，建议用户直接参考N32H47x_48x系列驱动程序。

n32g45x_spi.h

```
                        This parameter can be a value of @ref I2S_Clock_Polarity */
} I2S_InitType;

/**
  * @}
  */

/** @addtogroup SPI_Exported_Constants
  * @{
  */

#define IS_SPI_PERIPH(PERIPH) (((PERIPH) == SPI1) || ((PERIPH) == SPI2) || ((PERIPH) == SPI3))

#define IS_SPI_2OR3_PERIPH(PERIPH) (((PERIPH) == SPI2) || ((PERIPH) == SPI3))

/** @addtogroup SPI_data_direction
  * @{
  */

#define SPI_DIR_DOUBLELINE_FULLDUPLEX ((uint16_t)0x0000)
#define SPI_DIR_DOUBLELINE_RONLY      ((uint16_t)0x0400)
#define SPI_DIR_SINGLELINE_RX         ((uint16_t)0x8000)
#define SPI_DIR_SINGLELINE_TX         ((uint16_t)0xC000)
#define IS_SPI_DIR_MODE(MODE)
    (((MODE) == SPI_DIR_DOUBLELINE_FULLDUPLEX) || ((MODE) == SPI_DIR_DOUBLELINE_RONLY)
     || ((MODE) == SPI_DIR_SINGLELINE_RX) || ((MODE) == SPI_DIR_SINGLELINE_TX))
/**
  * @}
  */

/** @addtogroup SPI_mode
  * @{
  */

#define SPI_MODE_MASTER    ((uint16_t)0x0104)
#define SPI_MODE_SLAVE     ((uint16_t)0x0000)
#define IS_SPI_MODE(MODE) (((MODE) == SPI_MODE_MASTER) || ((MODE) == SPI_MODE_SLAVE))
/**
  * @}
  */

/** @addtogroup SPI_data_size
  * @{
  */

#define SPI_DATA_SIZE_16BITS       ((uint16_t)0x0800)
#define SPI_DATA_SIZE_8BITS        ((uint16_t)0x0000)
#define IS_SPI_DATASIZE(DATASIZE) (((DATASIZE) == SPI_DATA_SIZE_16BITS) || ((DATASIZE) == SPI_DATA_SIZE_8
/**
```

n32h47x_48x_spi.h

```
    uint16_t CLKPOL; /*!< Specifies the idle state of the I2S clock.
                          This parameter can be a value of @ref I2S_Clock_Polarity */
} I2S_InitType;


/** SPI_Exported_Constants **/

#define IS_SPI_PERIPH(PERIPH) (((PERIPH) == SPI1) || \
                               ((PERIPH) == SPI2) || \
                               ((PERIPH) == SPI3) || \
                               ((PERIPH) == SPI4) || \
                               ((PERIPH) == SPI5) || \
                               ((PERIPH) == SPI6))

#define IS_SPI_2OR3_PERIPH(PERIPH) (((PERIPH) == SPI2) || ((PERIPH) == SPI3))

/** SPI_data_direction **/

#define SPI_DIR_DOUBLELINE_FULLDUPLEX ((uint16_t)0x0000U)
#define SPI_DIR_DOUBLELINE_RONLY      ((uint16_t)0x2000U)
#define SPI_DIR_SINGLELINE_RX         ((uint16_t)0x8000U)
#define SPI_DIR_SINGLELINE_TX         ((uint16_t)0xC000U)
#define IS_SPI_DIR_MODE(MODE)
    (((MODE) == SPI_DIR_DOUBLELINE_FULLDUPLEX) || ((MODE) == SPI_DIR_DOUBLELINE_RONLY)
     || ((MODE) == SPI_DIR_SINGLELINE_RX) || ((MODE) == SPI_DIR_SINGLELINE_TX))

/** SPI_mode **/

#define SPI_MODE_MASTER    ((uint16_t)0x0840U)
#define SPI_MODE_SLAVE     ((uint16_t)0x0000U)
#define IS_SPI_MODE(MODE) (((MODE) == SPI_MODE_MASTER) || ((MODE) == SPI_MODE_SLAVE))


/** SPI_data_size **/

#define SPI_DATA_SIZE_16BITS       ((uint16_t)0x0100U)
#define SPI_DATA_SIZE_8BITS        ((uint16_t)0x0000U)
#define IS_SPI_DATASIZE(DATASIZE) (((DATASIZE) == SPI_DATA_SIZE_16BITS) || ((DATASIZE) == SPI_


/** SPI_Clock_Polarity **/

#define SPI_CLKPOL_LOW      ((uint16_t)0x0000U)
#define SPI_CLKPOL_HIGH     ((uint16_t)0x0010U)
#define IS_SPI_CLKPOL(CPOL) (((CPOL) == SPI_CLKPOL_LOW) || ((CPOL) == SPI_CLKPOL_HIGH))


/** SPI_Clock_Phase **/
```

和N32G45x系列相比，N32H47x_48x系列SPI增支持FIFO功能，所以N32H47x_48x系列开发套件中SPI模块增加了FIFO功能相关的Demo

4.3.6 I2C

下表是N32G45x系列和N32H47x_48x系列在I2C模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用I2C模块驱动API函数可以依照下表做替换：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
将 I2Cx 外围寄存器初始化为默认值	void I2C_DeInit(I2C_Module* I2Cx);	void I2C_DeInit(const I2C_Module* I2Cx);	是	无
初始化 I2Cx 外设	void I2C_Init(I2C_Module* I2Cx, I2C_InitType* I2C_InitStruct);	void I2C_Init(I2C_Module* I2Cx,const I2C_InitType* I2C_InitStruct);	是	无
初始化 I2C_InitStruct 结构体	void I2C_InitStruct(I2C_InitType* I2C_InitStruct);	Void I2C_InitStruct(I2C_InitType* I2C_InitStruct);	是	无
使能或禁能 I2C	void I2C_Enable(I2C_Module* I2Cx, FunctionalState Cmd);	void I2C_Enable(I2C_Module* I2Cx, FunctionalState Cmd);	是	无
使能或禁能 I2C DMA	void I2C_EnableDMA(I2C_Module* I2Cx, FunctionalState Cmd)	void I2C_EnableDMA(I2C_Module* I2Cx, FunctionalState Cmd);	是	无
指定下一次 DMA 传输是否为最后一次	void I2C_EnableDmaLastSend(I2C_Module* I2Cx, FunctionalState Cmd);	void I2C_EnableDmaLastSend(I2C_Module* I2Cx, FunctionalState Cmd);	是	无
I2C 产生起始位	void I2C_GenerateStart(I2C_Module* I2Cx, FunctionalState Cmd);	void I2C_GenerateStart(I2C_Module* I2Cx, FunctionalState Cmd);	是	无
I2C 产生停止位	void I2C_GenerateStop(I2C_Module* I2Cx, FunctionalState Cmd);	void I2C_GenerateStop(I2C_Module* I2Cx, FunctionalState Cmd);	是	无
启用或禁用指定的 I2C ACK 功能。	void I2C_ConfigAck(I2C_Module* I2Cx, FunctionalState Cmd);	void I2C_ConfigAck(I2C_Module* I2Cx, FunctionalState Cmd);	是	无

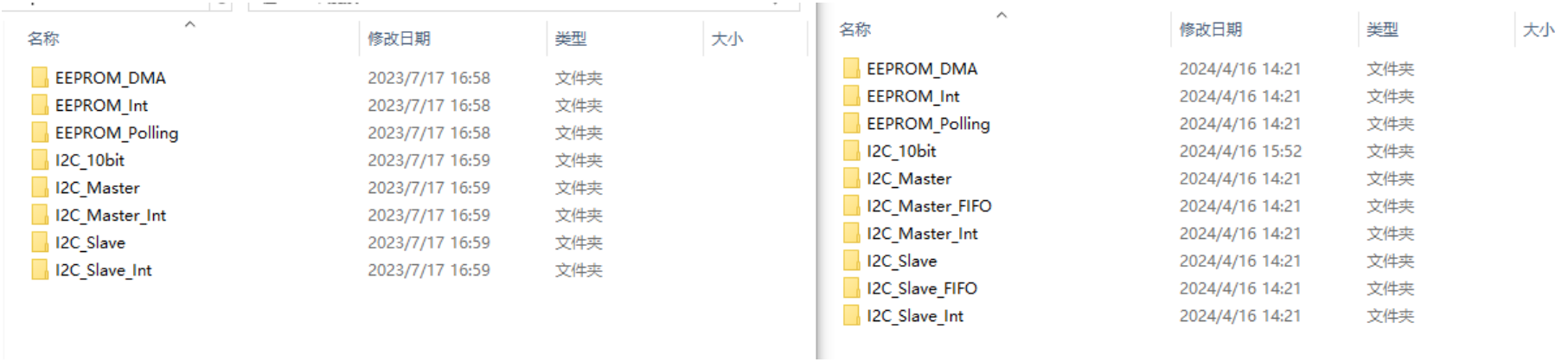
配置指定的 I2C 地址 2	void I2C_ConfigOwnAddr2(I2C_Module* I2Cx, uint8_t Address);	void I2C_ConfigOwnAddr2(I2C_Module* I2Cx, uint8_t Address);	是	无
启用或禁用指定的 I2C 双寻址模式	void I2C_EnableDualAddr(I2C_Module* I2Cx, FunctionalState Cmd);	void I2C_EnableDualAddr(I2C_Module* I2Cx, FunctionalState Cmd);	是	无
启用或禁用指定的 I2C 通用调用功能	void I2C_EnableGeneralCall(I2C_Module* I2Cx, FunctionalState Cmd);	void I2C_EnableGeneralCall(I2C_Module* I2Cx, FunctionalState Cmd);	是	无
启用或禁用指定的 I2C 中断	void I2C_ConfigInt(I2C_Module* I2Cx, uint16_t I2C_IT, FunctionalState Cmd);	void I2C_ConfigInt(I2C_Module* I2Cx, uint32_t I2C_IT, FunctionalState Cmd);	否	输入参数 I2C_IT 不一致： N32G45x 系列参数： I2C_INT_BUF I2C_INT_EVENT I2C_INT_ERR N32H47x_48x 系列参数： I2C_INT_BUF I2C_INT_EVENT I2C_INT_ERR I2C_INT_BUF I2C_INT_EVENT I2C_INT_ERR I2C_INT_FIFOF I2C_INT_FIFOE I2C_INT_FIFOHF I2C_INT_FIFOHE I2C_INT_FIFORDEIEN I2C_INT_FIFOWREIEN I2C_INT_DMAETOEIEN I2C_INT_FIFONE I2C_INT_FIFONF I2C_INT_SDATOLIEN I2C_INT_SCLTOHIEN I2C_INT_SCLTOLIEN
I2C 发送数据	void I2C_SendData(I2C_Module* I2Cx, uint8_t Data);	void I2C_SendData(I2C_Module* I2Cx, uint8_t Data);	是	无
I2C 接收数据	uint8_t I2C_RecvData(I2C_Module* I2Cx);	uint8_t I2C_RecvData(const I2C_Module* I2Cx);	是	无
传输地址字节以选择从设备	void I2C_SendAddr7bit(I2C_Module* I2Cx, uint8_t Address, uint8_t I2C_Direction);	void I2C_SendAddr7bit(I2C_Module* I2Cx, uint8_t Address, uint8_t I2C_Direction);	是	无
读取指定的 I2C 寄存器	uint16_t I2C_GetRegister(I2C_Module* I2Cx, uint8_t I2C_Register);	uint16_t I2C_GetRegister(const I2C_Module* I2Cx, uint8_t I2C_Register);	否	输入参数 I2C_Register 不一致： N32G45x 系列参数： I2C_INT_BUF I2C_INT_EVENT I2C_INT_ERR

				N32H47x_48x 系列参数在 N32G45x 系列参数基础上增加了 2 个参数：I2C_REG_BYTENUM I2C_REG_GFLTRCTRL
启用或禁用指定的 I2C 软件复位	void I2C_EnableSoftwareReset(I2C_Module* I2Cx, FunctionalState Cmd);	void I2C_EnableSoftwareReset(I2C_Module* I2Cx, FunctionalState Cmd);	是	无
在主接收机模式下选择指定的 I2C NACK 位置	void I2C_ConfigNackLocation(I2C_Module* I2Cx, uint16_t I2C_NACKPosition);	void I2C_ConfigNackLocation(I2C_Module* I2Cx, uint16_t I2C_NACKPosition);	是	无
驱动指定 I2C 的 SMBusAlert 引脚为高电平或低电平	void I2C_ConfigSmbusAlert(I2C_Module* I2Cx, uint16_t I2C_SMBusAlert);	void I2C_ConfigSmbusAlert(I2C_Module* I2Cx, uint16_t I2C_SMBusAlert);	是	无
启用或禁用指定的 I2C PEC 传输	void I2C_SendPEC(I2C_Module* I2Cx, FunctionalState Cmd);	void I2C_SendPEC(I2C_Module* I2Cx, FunctionalState Cmd);	是	无
选择指定的 I2C PEC 位置	void I2C_ConfigPecLocation(I2C_Module* I2Cx, uint16_t I2C_PECPosition);	void I2C_ConfigPecLocation(I2C_Module* I2Cx, uint16_t I2C_PECPosition);	是	无
启用或禁用传输字节的 PEC 值计算。	void I2C_ComputePec(I2C_Module* I2Cx, FunctionalState Cmd);	void I2C_ComputePec(I2C_Module* I2Cx, FunctionalState Cmd);	是	无
获取 PEC 值	uint8_t I2C_GetPec(I2C_Module* I2Cx);	uint8_t I2C_GetPec(const I2C_Module* I2Cx);	是	无
启用或禁用指定的 I2C ARP	I2C_EnableArp(I2C_Module* I2Cx, FunctionalState Cmd);	void I2C_EnableArp(I2C_Module* I2Cx, FunctionalState Cmd);	是	无
启用或禁用指定的 I2C 时钟拉伸	void I2C_EnableExtendClk(I2C_Module* I2Cx, FunctionalState Cmd);	void I2C_EnableExtendClk(I2C_Module* I2Cx, FunctionalState Cmd);	是	无
选择指定的 I2C 快速模式占空比	void I2C_ConfigFastModeDutyCycle(I2C_Module* I2Cx, uint16_t FmDutyCycle);	void I2C_ConfigFastModeDutyCycle(I2C_Module* I2Cx, uint16_t FmDutyCycle);	是	无
I2C check FIFO 事件	-	ErrorStatus I2C_CheckFifoEvent(I2C_Module* I2Cx, uint32_t I2C_FIFO_EVENT);	否	N32G45x 系列无该接口
I2C 获取上一个 FIFO 事件	-	uint32_t I2C_GetLastFifoEvent(I2C_Module* I2Cx);	否	N32G45x 系列无该接口
清除 I2C FIFO 标志位	-	void I2C_ClrFIFO(I2C_Module* I2Cx);	否	N32G45x 系列无该接口
启用或禁用 FIFO	-	void I2C_EnableFIFO(I2C_Module* I2Cx, FunctionalState Cmd);	否	N32G45x 系列无该接口
启用或禁用 DMA FIFO	-	void I2C_EnableDMAFIFO(I2C_Module* I2Cx, FunctionalState Cmd);	否	N32G45x 系列无该接口
启用或禁用主机接收字节计数功能	-	void I2C_EnableBYTENUM(I2C_Module* I2Cx, FunctionalState Cmd);	否	N32G45x 系列无该接口
配置 master 完成发送所需数据字节后的最后状态	-	void I2C_SetByteNumLastStartStop(I2C_Module* I2Cx, uint16_t LastStatus);	否	N32G45x 系列无该接口

设置 I2C 外围设备接收器将接收的数据字节数	-	void I2C_SetReceivedDataBytesNum(I2C_Module* I2Cx, uint16_t Number_Of_bytes);	否	N32G45x 系列无该接口
设置 I2C 的 Tx FIFO 阈值	-	void I2C_SetTxFifoThreshold(I2C_Module* I2Cx, uint8_t TxFifoThreshold);	否	N32G45x 系列无该接口
设置 I2C 的 Rx FIFO 阈值	-	void I2C_SetRxFifoThreshold(I2C_Module* I2Cx, uint8_t RxFifoThreshold);	否	N32G45x 系列无该接口
设置 I2C 的 FIFO DATA	-	void I2C_SetFIFODAT(I2C_Module* I2Cx, uint32_t I2C_BYTENUM);	否	N32G45x 系列无该接口
获取 I2C 的 FIFO DATA	-	uint8_t I2C_GetFIFODAT(const I2C_Module* I2Cx);	否	N32G45x 系列无该接口
启用或禁用最低超时时间	-	void I2C_EnableLowTimeout(I2C_Module* I2Cx, FunctionalState Cmd);	否	N32G45x 系列无该接口
启用或禁用最高超时时间	-	void I2C_EnableHighTimeout(I2C_Module* I2Cx, FunctionalState Cmd);	否	N32G45x 系列无该接口
启用或禁用 SCL 模拟滤波器	-	void I2C_EnableSCLAnalogFilter(I2C_Module* I2Cx, FunctionalState Cmd);	否	N32G45x 系列无该接口
启用或禁用 SDA 模拟滤波器	-	void I2C_EnableSDAAnalogFilter(I2C_Module* I2Cx, FunctionalState Cmd);	否	N32G45x 系列无该接口
SCL 模拟滤波器宽度选择	-	void I2C_SetSCLAnalogFilterWidth(I2C_Module* I2Cx, uint32_t width);	否	N32G45x 系列无该接口
SDA 模拟滤波器宽度选择	-	void I2C_SetSDAAnalogFilterWidth(I2C_Module* I2Cx, uint32_t width);	否	N32G45x 系列无该接口
SDA 数字滤波器宽度选择	-	void I2C_SetSDADigitalFilterWidth(I2C_Module* I2Cx, uint32_t width);	否	N32G45x 系列无该接口
SCL 数字滤波器宽度选择	-	void I2C_SetSCLDigitalFilterWidth(I2C_Module* I2Cx, uint32_t width);	否	N32G45x 系列无该接口
检查最后一个 I2Cx 事件是否等于作为参数传递的事件	-	ErrorStatus I2C_CheckEvent(const I2C_Module* I2Cx, uint32_t I2C_EVENT);	否	N32G45x 系列无该接口
获取最后一个 I2Cx 事件	-	uint32_t I2C_GetLastEvent(const I2C_Module* I2Cx);	否	N32G45x 系列无该接口
检查指定的 I2C 标志是否已设置。	-	FlagStatus I2C_GetFlag(const I2C_Module* I2Cx, uint32_t I2C_FLAG);	否	N32G45x 系列无该接口
清除指定的 I2C 标志	-	void I2C_ClrFlag(I2C_Module* I2Cx, uint32_t I2C_FLAG);	否	N32G45x 系列无该接口
检查指定的 I2C 中断是否已发生	-	INTStatus I2C_GetIntStatus(const I2C_Module* I2Cx, uint32_t I2C_IT);	否	N32G45x 系列无该接口
清除 I2Cx 的中断挂起位	-	void I2C_ClrIntPendingBit(I2C_Module* I2Cx, uint32_t I2C_IT);	否	N32G45x 系列无该接口

N32H47x_48x系列和N32G45x系列I2C使用流程基本一致，但是他们的宏定义差别很大，建议用户直接参考N32H47x_48x系列驱动程序。
另外，N32H47x_48x系列比N32G45x系列I2C增加了FIFO收发数据的功能，所以Demo中增加了FIFO相关的Demo。如下图所示N32G45x系列

（左）和N32H47x_48x系列（右）：



名称	修改日期	类型	大小
EEPROM_DMA	2023/7/17 16:58	文件夹	
EEPROM_Int	2023/7/17 16:58	文件夹	
EEPROM_Polling	2023/7/17 16:58	文件夹	
I2C_10bit	2023/7/17 16:59	文件夹	
I2C_Master	2023/7/17 16:59	文件夹	
I2C_Master_Int	2023/7/17 16:59	文件夹	
I2C_Slave	2023/7/17 16:59	文件夹	
I2C_Slave_Int	2023/7/17 16:59	文件夹	

名称	修改日期	类型	大小
EEPROM_DMA	2024/4/16 14:21	文件夹	
EEPROM_Int	2024/4/16 14:21	文件夹	
EEPROM_Polling	2024/4/16 14:21	文件夹	
I2C_10bit	2024/4/16 15:52	文件夹	
I2C_Master	2024/4/16 14:21	文件夹	
I2C_Master_FIFO	2024/4/16 14:21	文件夹	
I2C_Master_Int	2024/4/16 14:21	文件夹	
I2C_Slave	2024/4/16 14:21	文件夹	
I2C_Slave_FIFO	2024/4/16 14:21	文件夹	
I2C_Slave_Int	2024/4/16 14:21	文件夹	

4.3.7 ALGO

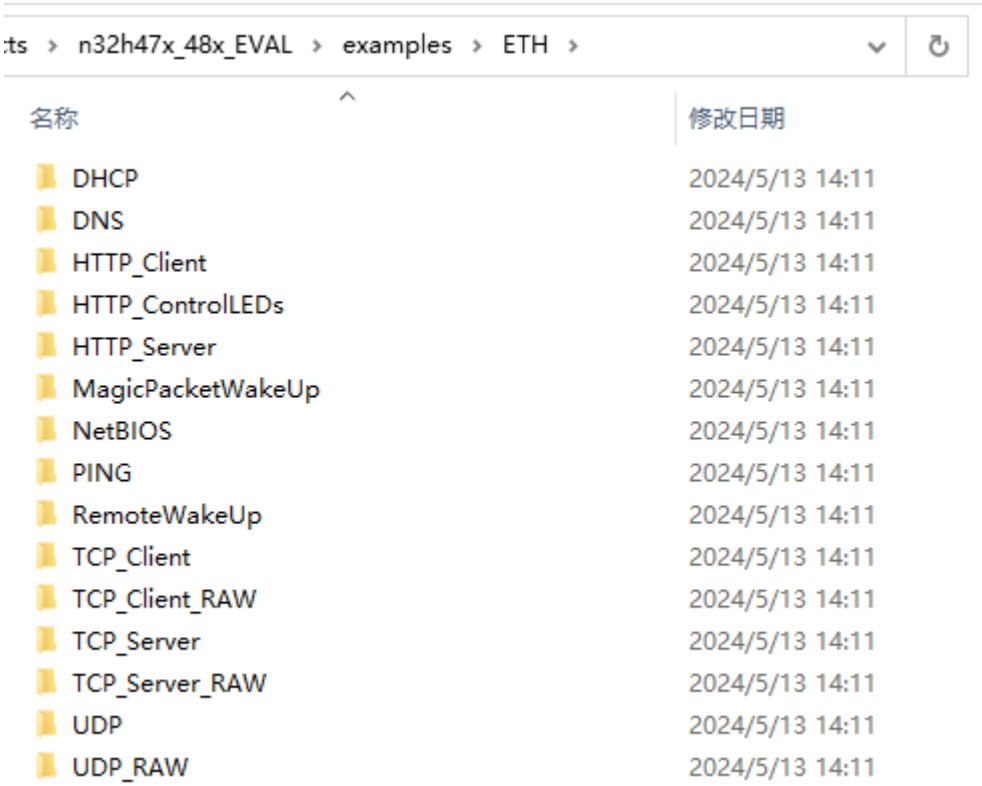
ALGO模块驱动已经封装成库，使用中调用接口函数即可。N32H47x_48x系列和N32G45x系列都提供了AES、DES、HASH、Rand、MD5算法的使用Demo。除此之外，N32H47x_48x系列还提供了SM4、SM3的调用Demo。建议用户直接参考N32H47x_48x系列开发套件的Demo程序。
下图是N32H47x_48x系列（左）和N32G45x系列（右）ALGO模块的“main”函数示例工程代码对比：

```c
/**
 *\*\name    main.
 *\*\fun     Main program.
 *\*\param   none
 *\*\return  none
 **/
 int main(void)
{
    log_init();
    log_info("----------------------\nAlgorithm demo start.\n");


    SM4_test();
    AES_128_test();
    AES_192_test();
    AES_256_test();
    DES_test();
    TDES_2Key_test();
    TDES_3Key_test();
    HASH_test();
    SM3_test();
    MD5_test();
    TestRand();

    log_info("\r\nALGO demo all test OK!\r\n");
    while (1)
        ;
}
```

```c
}

/**
 * @brief  main function.
 */
int main(void)
{
    log_init();
    log_info("----------------------\nAlgorithm demo start.\n");

    // RNG test
    TestRand();

    // HASH test
    TestMD5();
    TestSHA1();
    TestSHA224();
    TestSHA256();

    // Cryptogram algorithm
    TestDES();
    TestAES();

    log_info("\r\nALGO demo all test OK!\r\n");
    while (1)
        ;
}
```

4.3.8 ETH

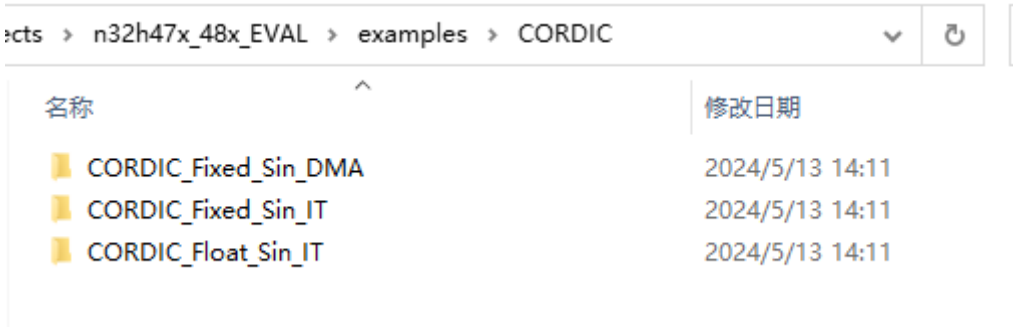
N32H47x_48x系列的ETH模块IP有升级，但以太网功能无任何影响。其中，ETH寄存器与N32G45x系列相比差异较大，故对应驱动函数存在较大差异，包括编程风格等也有所不同。另外，在开发ETH时，除驱动.c/.h文件外还需要其他必要文件，这些文件与N32G45x系列相比完全不同，甚至N32G45x系列无这些文件。所以在开发时建议用户参考SDK中提供的示例，如下图所示：



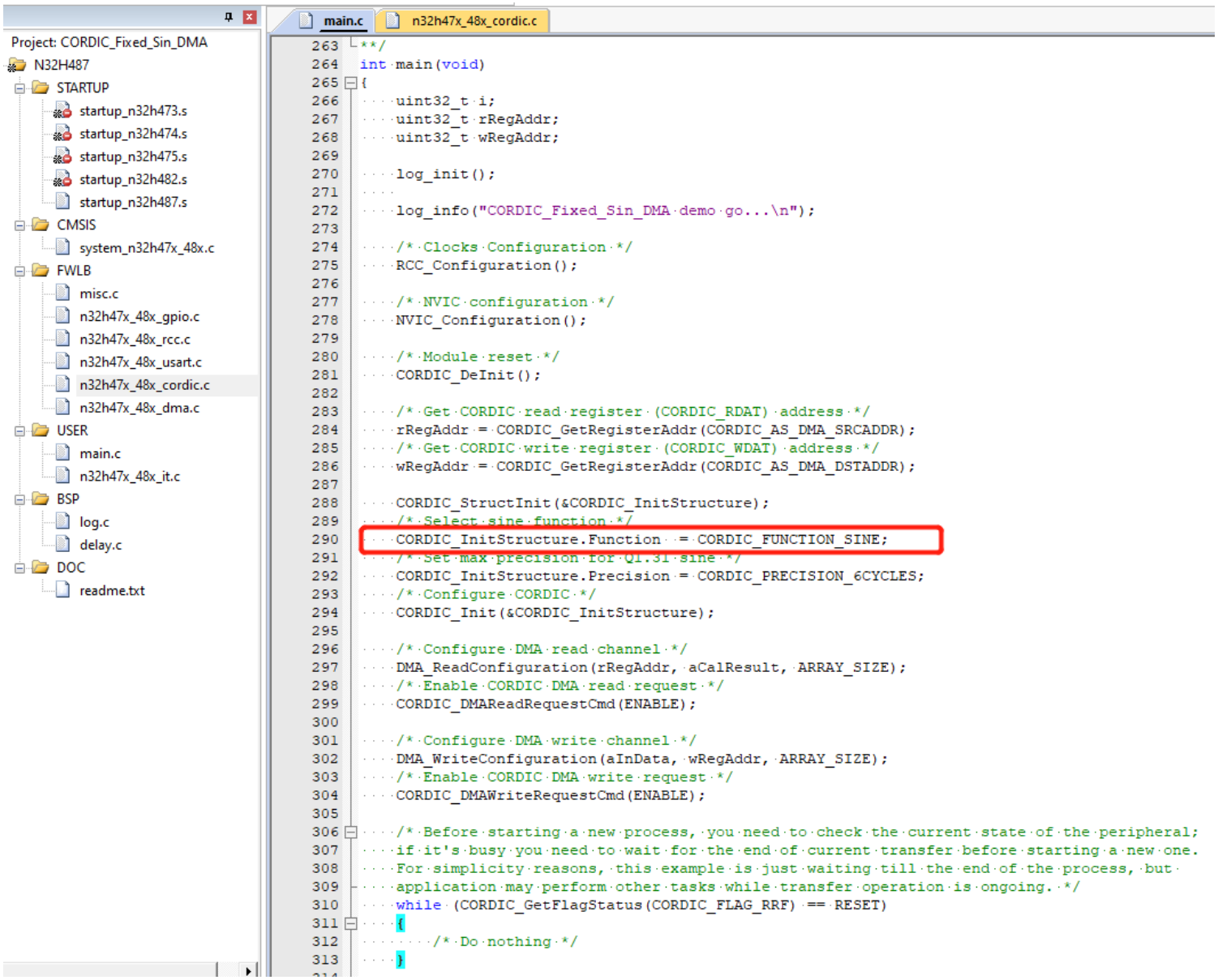
目前提供的示例中，包含有操作系统和无操作系统两种类型（例程NetBIOS、PING、TCP_Client_RAW、TCP_Server_RAW、UDP_RAW不带操作系统，其他均基于FreeRTOS v202212.01版），用户可按需参考相应例程。其他协议例程还在逐步完善中，将随后续版本新增发布。
特别提示：参考示例时，建议关注对应例程的readme文件中的注意事项。

4.3.9 CORDIC

相较于N32G45x系列，CORDIC为新增模块，该模块用于数学函数运算（主要是三角函数）。SDK中以正弦函数为例，提供了“定点输入+DMA方式”、“定点输入+中断方式”和“浮点输入+中断方式”三种场景下的示例工程，如下图所示：

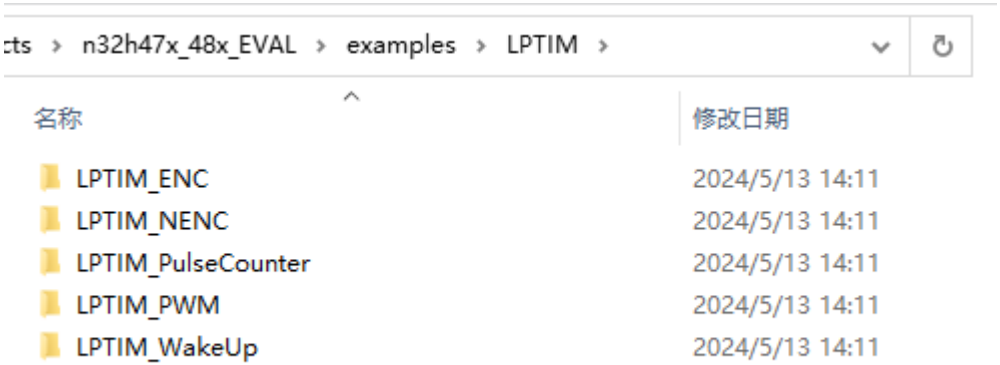


用户可根据输入类型和工作方式参考对应场景的例程，并修改为所需的数学函数，例如修改下图例程中的红框部分。


Project: CORDIC_Fixed_Sin_DMA
N32H487
STARTUP
startup_n32h473.s
startup_n32h474.s
startup_n32h475.s
startup_n32h482.s
startup_n32h487.s
CMSIS
system_n32h47x_48x.c
FWLB
misc.c
n32h47x_48x_gpio.c
n32h47x_48x_rcc.c
n32h47x_48x_usart.c
n32h47x_48x_cordic.c
n32h47x_48x_dma.c
USER
main.c
n32h47x_48x_it.c
BSP
log.c
delay.c
DOC
readme.txt
main.c
n32h47x_48x_cordic.c
**/
int main(void)
{
uint32_t i;
uint32_t rRegAddr;
uint32_t wRegAddr;
log_init();
log_info("CORDIC_Fixed_Sin_DMA demo go...\n");
/* Clocks Configuration */
RCC_Configuration();
/* NVIC configuration */
NVIC_Configuration();
/* Module reset */
CORDIC_DeInit();
/* Get CORDIC read register (CORDIC_RDAT) address */
rRegAddr = CORDIC_GetRegisterAddr(CORDIC_AS_DMA_SRCADDR);
/* Get CORDIC write register (CORDIC_WDAT) address */
wRegAddr = CORDIC_GetRegisterAddr(CORDIC_AS_DMA_DSTADDR);
CORDIC_StructInit(&CORDIC_InitStructure);
/* Select sine function */
CORDIC_InitStructure.Function = CORDIC_FUNCTION_SINE;
/* Set max precision for Q1.31 sine */
CORDIC_InitStructure.Precision = CORDIC_PRECISION_6CYCLES;
/* Configure CORDIC */
CORDIC_Init(&CORDIC_InitStructure);
/* Configure DMA read channel */
DMA_ReadConfiguration(rRegAddr, aCalResult, ARRAY_SIZE);
/* Enable CORDIC DMA read request */
CORDIC_DMAReadRequestCmd(ENABLE);
/* Configure DMA write channel */
DMA_WriteConfiguration(aInData, wRegAddr, ARRAY_SIZE);
/* Enable CORDIC DMA write request */
CORDIC_DMAWriteRequestCmd(ENABLE);
/* Before starting a new process, you need to check the current state of the peripheral;
if it's busy you need to wait for the end of current transfer before starting a new one.
For simplicity reasons, this example is just waiting till the end of the process, but
application may perform other tasks while transfer operation is ongoing. */
while (CORDIC_GetFlagStatus(CORDIC_FLAG_RRF) == RESET)
{
/* Do nothing */
}

4.3.10 LPTIM

相较于N32G45x系列，LPTIM为新增模块，该模块定时器可工作于所有功耗模式，并具备唤醒能力（除VBAT模式外）。SDK中提供了如下图所示的示例工程：



如需要计算正交编码模式下的信号边沿数，可参考“LPTIM_ENC”例程；
如需要计算非正交编码模式下的信号边沿数，可参考“LPTIM_NENC”例程；
如需要计数脉冲个数，可参考“LPTIM_PulseCounter”例程；
如需要生成PWM波形，可参考“LPTIM_PWM”例程；
如需要工作于并唤醒低功耗模式，可参考“LPTIM_WakeUp”例程，如何进入除该示例外的其他低功耗模式，可参考PWR模块的相应例程。

4.3.11 GPIO

GPIO的基本配置与端口重映射差异较大。N32G45x系列和N32H47x_48x系列GPIO模块的驱动API函数对比详见下表，替换驱动.c/.h文件后，应用代码调用GPIO模块驱动API函数可以依照下表做替换：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
GPIO 复位	void GPIO_DeInit(GPIO_Module* GPIOx)	void GPIO_DeInit(GPIO_Module* GPIOx)	是	输入参数 GPIOx 取值范围不同： N32H47x_48x 系列增加 GPIOH 选项
复位端口为默认配置	-	void GPIO_DeInitPin(GPIO_Module* GPIOx, uint32_t Pin)	否	N32G45x 系列无此 API
AFIO 复位	void GPIO_AFIOInitDefault(void)	void GPIO_AFIOInitDefault(void)	是	无

GPIO 初始化	void GPIO_InitPeripheral(GPIO_Module* GPIOx, GPIO_InitType* GPIO_InitStruct)	void GPIO_InitPeripheral(GPIO_Module* GPIOx, GPIO_InitType * GPIO_InitStruct)	否	函数名与输入参数相同，但两个系列 GPIO 的配置方式不同，输入参数指向的结构体定义差异较大，请参照用户手册及 SDK： N32G45x 系列结构体：Pin、GPIO_Speed、GPIO_Mode N32H47x_48x 系列结构体:Pin、GPIO_Mode、GPIO_Pull、GPIO_Slew_Rate、GPIO_Current、GPIO_Alternate
GPIO 结构体初始化	void GPIO_InitStruct(GPIO_InitType* GPIO_InitStruct)	void GPIO_InitStruct(GPIO_InitType* InitStruct)	是	输入参数指向的结构体定义差异较大
获取单个端口输入状态	uint8_t GPIO_ReadInputDataBit(GPIO_Module* GPIOx, uint16_t Pin)	uint8_t GPIO_ReadInputDataBit(GPIO_Module* GPIOx, uint16_t Pin)	是	输入参数 GPIOx 取值范围不同： N32H47x_48x 系列增加 GPIOH 选项
获取端口组输入状态	uint16_t GPIO_ReadInputData(GPIO_Module* GPIOx)	uint16_t GPIO_ReadInputData(GPIO_Module* GPIOx)	是	输入参数 GPIOx 取值范围不同： N32H47x_48x 系列增加 GPIOH 选项
获取单个端口输出状态	uint8_t GPIO_ReadOutputDataBit(GPIO_Module* GPIOx, uint16_t Pin)	uint8_t GPIO_ReadOutputDataBit(GPIO_Module* GPIOx, uint16_t Pin)	是	输入参数 GPIOx 取值范围不同： N32H47x_48x 系列增加 GPIOH 选项
获取端口组输出状态	uint16_t GPIO_ReadOutputData(GPIO_Module* GPIOx)	uint16_t GPIO_ReadOutputData(GPIO_Module* GPIOx)	是	输入参数 GPIOx 取值范围不同： N32H47x_48x 系列增加 GPIOH 选项
配置端口输出高电平	void GPIO_SetBits(GPIO_Module* GPIOx, uint16_t Pin)	void GPIO_SetBits(GPIO_Module* GPIOx, uint16_t Pin)	是	输入参数 GPIOx 取值范围不同： N32H47x_48x 系列增加 GPIOH 选项
配置端口输出高电平	void GPIO_SetBitsHigh16(GPIO_Module* GPIOx, uint32_t Pin)	-	否	N32H47x_48x 系列删除
配置端口输出低电平	void GPIO_ResetBits(GPIO_Module* GPIOx, uint16_t Pin)	void GPIO_ResetBits(GPIO_Module* GPIOx, uint16_t Pin)	是	输入参数 GPIOx 取值范围不同： N32H47x_48x 系列增加 GPIOH 选项
配置端口输出状态	void GPIO_WriteBit(GPIO_Module* GPIOx, uint16_t Pin, Bit_OperateType BitCmd)	void GPIO_WriteBits(GPIO_Module* GPIOx, uint16_t Pin, Bit_OperateType BitCmd)	是	输入参数 GPIOx 取值范围不同： N32H47x_48x 系列增加 GPIOH 选项
配置端口组输出状态	void GPIO_Write(GPIO_Module* GPIOx, uint16_t PortVal)	void GPIO_Write(GPIO_Module* GPIOx, uint16_t data_value)	是	输入参数 GPIOx 取值范围不同： N32H47x_48x 系列增加 GPIOH 选项
端口输出状态翻转	-	void GPIO_TogglePin(GPIO_Module *GPIOx, uint16_t Pin)	否	N32H47x_48x 系列新增函数
锁定端口寄存器	void GPIO_ConfigPinLock(GPIO_Module* GPIOx, uint16_t Pin)	void GPIO_ConfigPinLock(GPIO_Module* GPIOx, uint16_t Pin)	是	输入参数 GPIOx 取值范围不同： N32H47x_48x 系列增加 GPIOH 选项
配置 EVENT 输出端口	void GPIO_ConfigEventOutput(uint8_t PortSource, uint8_t PinSource)	-	否	N32H47x_48x 系列 EVENT 输出支持管脚全映射，EVENT 端口改在 GPIO 初始化时配置
使能 EVENT 输出	void GPIO_CtrlEventOutput(FunctionalState Cmd)	-	否	N32H47x_48x 系列不需要使能

配置外部中断端口	void GPIO_ConfigEXTILine(uint8_t PortSource, uint8_t PinSource)	void GPIO_ConfigEXTILine(uint8_t LineSource, uint8_t PortSource, uint8_t PinSource)	否	N32H47x_48x 系列 EXTI 输入支持管脚全映射，新增输入参数 LineSource: EXTI_LINE_SOURCE0～15
配置外设端口重映射	void GPIO_ConfigPinRemap(uint32_t RmpPin, FunctionalState Cmd)	void GPIO_ConfigPinRemap(uint8_t PortSource, uint8_t PinSource, uint32_t AlternateFunction)	否	N32G45x 系列必须按照外设端口组来重映射，而 N32H47x_48x 系列每个端口可分别指定重映射，差异很大，请参照用户手册及 SDK
配置 SPI 空闲时 NSS 端口状态	-	void AFIO_Config_SPI_NSS_Mode(uint32_t AFIO_SPIx_NSS, uint32_t NSS_Mode)	否	N32H47x_48x 系列新增函数
配置端口滤波周期	-	void AFIO_ConfigIOFilter(uint32_t Filter_Cycle)	否	N32H47x_48x 系列新增函数
配置 FEMC 模块 NADV 端口	-	void AFIO_Config_FEMC_NADV_Pin(uint32_t NADV_Pin)	否	N32H47x_48x 系列新增函数
配置 EXTI 模拟滤波	-	void AFIO_Config_EXTI_Filter(uint32_t EXTI_Filter)	否	N32H47x_48x 系列新增函数
配置 xSPI XIP 模式下的数据大小端模式	-	void AFIO_Config_XSPI_XIP_BigEndian(uint32_t Endian)	否	N32H47x_48x 系列新增函数
配置 xSPI 半双工模式	-	void AFIO_Config_XSPI_HalfDuplexMode(uint32_t HalfDuplex)	否	N32H47x_48x 系列新增函数
配置 xSPI 双四线模式	-	void AFIO_Config_XSPI_DualQuadMode(uint32_t DualQuad)	否	N32H47x_48x 系列新增函数
配置 ETH 接口模式	void GPIO_ETH_ConfigMediaInterface(uint32_t ETH_ConfigSel)	void AFIO_Config_ETH_Mode(uint32_t ETH_Mode)	否	函数名和输入参数宏定义增均不同： N32G45x 系列输入参数 ETH_ConfigSel： GPIO_ETH_RMII_CFG GPIO_ETH_MII_CFG N32H47x_48x 系列输入参数 ETH_Mode: AFIO_ETH_MODE_RMII AFIO_ETH_MODE_MII
配置 xSPI NSS 端口输入功能	-	void AFIO_Config_XSPI_NSS_Input(uint32_t InputEnable, uint32_t InputPin)	否	N32H47x_48x 系列新增函数
配置 xSPI RXDS 信号采样延迟时间	-	void AFIO_Config_XSPI_RXDS_SampleDelay(uint32_t Delay)	否	N32H47x_48x 系列新增函数

配置 xSPI 非 XIP 模式下读数据的大小端模式	-	void AFIO_Config_XSPI_BigEndian_Read(uint32_t Endian)	否	N32H47x_48x 系列新增函数
配置 xSPI 非 XIP 模式下写数据的大小端模式	-	void AFIO_Config_XSPI_BigEndian_Write(uint32_t Endian)	否	N32H47x_48x 系列新增函数
配置 xSPI 时间扩展功能	-	void AFIO_Config_XSPI_TimeExtension(uint32_t Extension)	否	N32H47x_48x 系列新增函数
配置 xSPI 高电平时间	-	void AFIO_Config_XSPI_NSS_HighTime(uint32_t clk)	否	N32H47x_48x 系列新增函数
配置端口 5V 兼容功能	-	void AFIO_ConfigPinTol5V(GPIO_Module* GPIOx, uint32_t Pin,FunctionalState cmd)	否	N32H47x_48x 系列新增函数
配置端口数字滤波功能	-	void AFIO_ConfigPinFilter(GPIO_Module* GPIOx, uint32_t Pin,FunctionalState cmd)	否	N32H47x_48x 系列新增函数
配置 GPIOC 寄存器读延迟功能	-	void AFIO_Config_GPIOC_ReadRegDelay(FunctionalState cmd)	否	N32H47x_48x 系列新增函数
配置 SHRTIM 外部事件端口	-	void AFIO_Config_SHRTIM_EXEV_Pin(uint8_t EXEV_Line, uint8_t PortSource, uint8_t PinSource)	否	N32H47x_48x 系列新增函数
配置 EMC 功能	-	void AFIO_Config_EMC_Funtion(uint32_t EMC_fun, FunctionalState cmd)	否	N32H47x_48x 系列新增函数
配置 EMC 计数值	-	void AFIO_Set_EMC_Cnt(uint32_t cnt)	否	N32H47x_48x 系列新增函数
获取 EMC 计数值	-	uint32_t AFIO_Get_EMC_Cnt(void)	否	N32H47x_48x 系列新增函数

N32H47x_48x系列GPIO模块的AFIO部分新增功能比较多，主要差异如下：

1) GPIO 端口全局数字滤波周期配置，每个端口数字滤波功能可单独启用/禁用
2) EXTI 外部中断可配置为任意管脚
3) EXTI 模拟滤波可配置
4) 部分管脚可配置 5V 兼容
5) FEMC NADV 管脚连接可配置

6) 不再支持 ADC 规则/注入转换触发重映射，改为在 ADC 模块中配置

7) 增加 xSPI 模块双四线模式、半双工模式、数据大小端模式、以及主机模式下 NSS 输入控制

8) 增加 SHRTIM 外部事件管脚配置，支持管脚全映射

4.3.12 EXTI

N32G45x系列和N32H47x_48x系列EXTI模块基本相同，寄存器定义有调整。同时，外部中断线由22条增加至32条，且外部中断支持管脚全映射（在AFIO中配置）。驱动API函数对比详见下表，替换驱动.c/.h文件后，应用代码调用EXTI模块驱动API函数可以依照下表做替换：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
EXTI 复位	void EXTI_DeInit(void)	void EXTI_DeInit(void)	是	无
EXTI 初始化	void EXTI_InitPeripheral(EXTI_InitType* EXTI_InitStruct)	void EXTI_InitPeripheral(EXTI_InitType* EXTI_InitStruct)	是	无
EXTI 结构体初始化	void EXTI_InitStruct(EXTI_InitType* EXTI_InitStruct)	void EXTI_InitStruct(EXTI_InitType* InitStruct)	是	无
触发 EXTI 软件中断	void EXTI_TriggerSWInt(uint32_t EXTI_Line)	void EXTI_TriggerSWInt(uint32_t EXTI_Line)	是	输入参数 EXTI_Line 取值范围不同： N32G45x 系列：EXTI_LINE0～EXTI_LINE21 N32H47x_48x 系列:EXTI_LINE0～EXTI_LINE31
获取 EXTI 状态标志	FlagStatus EXTI_GetStatusFlag(uint32_t EXTI_Line)	FlagStatus EXTI_GetStatusFlag(uint32_t EXTI_Line)	是	输入参数 EXTI_Line 取值范围不同： N32G45x 系列：EXTI_LINE0～EXTI_LINE21 N32H47x_48x 系列:EXTI_LINE0～EXTI_LINE31
清除 EXTI 状态标志	void EXTI_ClrStatusFlag(uint32_t EXTI_Line)	void EXTI_ClrStatusFlag(uint32_t EXTI_Line)	是	输入参数 EXTI_Line 取值范围不同： N32G45x 系列：EXTI_LINE0～EXTI_LINE21 N32H47x_48x 系列:EXTI_LINE0～EXTI_LINE31
获取 EXTI 中断标志	INTStatus EXTI_GetITStatus(uint32_t EXTI_Line)	INTStatus EXTI_GetITStatus(uint32_t EXTI_Line)	是	输入参数 EXTI_Line 取值范围不同： N32G45x 系列：EXTI_LINE0～EXTI_LINE21

				N32H47x_48x 系列:EXTI_LINE0～EXTI_LINE31
清除 EXTI 中断标志	void EXTI_ClrITPendBit(uint32_t EXTI_Line)	void EXTI_ClrITPendBit(uint32_t EXTI_Line)	是	输入参数 EXTI_Line 取值范围不同： N32G45x 系列：EXTI_LINE0～EXTI_LINE21 N32H47x_48x 系列:EXTI_LINE0～EXTI_LINE31
选择时间戳触发源	void EXTI_RTCTimeStampSel(uint32_t EXTI_TSSEL_Line)	void EXTI_RTCTimeStampSel(uint32_t EXTI_TSSEL_Line)	是	无

4.3.13 SDIO

N32G45x系列和N32H47x_48x系列SDIO模块基本相同，寄存器定义有调整。驱动API函数对比详见下表，替换驱动.c/.h文件后，应用代码调用SDIO模块驱动API函数可以依照下表做替换：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
SDIO 复位	void SDIO_DeInit(void)	void SDIO_DeInit(void)	是	无
SDIO 初始化	void SDIO_Init(SDIO_InitType* SDIO_InitStruct)	void SDIO_Init(SDIO_InitType* InitStruct)	否	初始化结构体成员 ClkDiv 取值范围有变化： N32G45x 系列：0x00～0xFF N32H47x_48x 系列：0x00～0x1FF
SDIO 结构休初始化	void SDIO_InitStruct(SDIO_InitType* SDIO_InitStruct)	void SDIO_InitStruct(SDIO_InitType* InitStruct)	否	初始化结构体成员 ClkDiv 取值范围有变化： N32G45x 系列：0x00～0xFF N32H47x_48x 系列：0x00～0x1FF
使能 SDIO 总线时钟输出	void SDIO_EnableClock(FunctionalState Cmd)	void SDIO_EnableClock(FunctionalState Cmd)	是	无
配置 SDIO 电源	void SDIO_SetPower(uint32_t SDIO_PowerState)	void SDIO_SetPower(uint32_t PowerState)	是	无
获取当前电源状态	uint32_t SDIO_GetPower(void)	uint32_t SDIO_GetPower(void)	否	返回值不同： N32G45x 系列：0x00、0x02、0x03 N32H47x_48x 系列： SDIO_POWER_OFF SDIO_POWER_UP SDIO_POWER_ON

配置 SDIO 中断	void SDIO_ConfigInt(uint32_t SDIO_IT, FunctionalState Cmd)	void SDIO_ConfigInt(uint32_t Int, FunctionalState Cmd)	否	第 1 个输入参数有变化： N32H47x_48x 系列不支持 SDIO_INT_CEATAF
启用/禁用 DMA	void SDIO_DMACmd(FunctionalState Cmd)	void SDIO_DMACmd(FunctionalState Cmd)	是	无
发送命令	void SDIO_SendCmd(SDIO_CmdInitType* SDIO_CmdInitStruct)	void SDIO_SendCmd(SDIO_CmdInitType* CmdInitStruct)	是	无
命令结构体初始化	void SDIO_InitCmdStruct(SDIO_CmdInitType* SDIO_CmdInitStruct)	void SDIO_InitCmdStruct(SDIO_CmdInitType* CmdInitStruct)	是	无
获取上次收到响应的命令索引	uint8_t SDIO_GetCmdResp(void)	uint8_t SDIO_GetCmdResp(void)	是	无
获取上次收到的响应	uint32_t SDIO_GetResp(uint32_t SDIO_RESP)	uint32_t SDIO_GetResp(uint32_t SDIO_RESP)	是	无
配置数据传输控制寄存器	void SDIO_ConfigData(SDIO_DataInitType* SDIO_DataInitStruct)	void SDIO_ConfigData(SDIO_DataInitType* DataInitStruct)	是	无
数据结构体初始化	void SDIO_InitDataStruct(SDIO_DataInitType* SDIO_DataInitStruct)	void SDIO_InitDataStruct(SDIO_DataInitType* SDIO_DataInitStruct)	是	无
获取未完成的传输数据量	uint32_t SDIO_GetDataCountValue(void)	uint32_t SDIO_GetDataCountValue(void)	是	无
从接收 FIFO 读取数据	uint32_t SDIO_ReadData(void)	uint32_t SDIO_ReadData(void)	是	无
写入数据到发送 FIFO	void SDIO_WriteData(uint32_t Data)	void SDIO_WriteData(uint32_t Data)	是	无
获取待写入或待读取的 FIFO 数据字数	uint32_t SDIO_GetFifoCounter(void)	uint32_t SDIO_GetFifoCounter(void)	是	无
启用读等待功能	void SDIO_EnableReadWait(FunctionalState Cmd)	void SDIO_EnableReadWait(FunctionalState Cmd)	是	无
开始/停止读等待	void SDIO_DisableReadWait(FunctionalState Cmd)	-	否	N32H47x_48x 系列拆分为两个函数 void SDIO_StopReadWait(void) void SDIO_StartReadWait(void)
停止读等待	-	void SDIO_StopReadWait(void)	否	N32H47x_48x 系列新增函数
开始读等待	-	void SDIO_StartReadWait(void)	否	N32H47x_48x 系列新增函数
配置读等待模式	void SDIO_EnableSdioReadWaitMode(uint32_t SDIO_ReadWaitMode)	void SDIO_EnableReadWaitMode(uint32_t ReadWaitMode)	是	无
配置 SD I/O 模式	void SDIO_EnableSdioOperation(FunctionalState Cmd)	void SDIO_EnableSdioOperation(FunctionalState Cmd)	是	无
配置 SD I/O 暂停命令	void SDIO_EnableSendSdioSuspend(FunctionalState Cmd)	void SDIO_EnableSendSdioSuspend(FunctionalState Cmd)	是	无
配置命令完成信号	void SDIO_EnableCommandCompletion(FunctionalState Cmd)	void SDIO_EnableCommandCompletion(FunctionalState Cmd)	是	无

配置 CE-ATA 设备中断	void SDIO_EnableCEATAInt(FunctionalState Cmd)	-	否	N32H47x_48x 不支持 CE-ATA 设备
配置 CE-ATA 命令发送	void SDIO_EnableSendCEATA(FunctionalState Cmd)	-	否	N32H47x_48x 不支持 CE-ATA 设备
获取 SDIO 状态标志	FlagStatus SDIO_GetFlag(uint32_t SDIO_FLAG)	FlagStatus SDIO_GetFlag(uint32_t Flag)	否	输入参数有变化： N32H47x_48x 系列不支持 SDIO_FLAG_CEATAF
获取 SDIO 中断标志	INTStatus SDIO_GetIntStatus(uint32_t SDIO_IT)	INTStatus SDIO_GetIntStatus(uint32_t Int)	否	输入参数有变化： N32H47x_48x 系列不支持 SDIO_INT_CEATAF
清除 SDIO 状态标志	void SDIO_ClrFlag(uint32_t SDIO_FLAG)	void SDIO_ClrFlag(uint32_t FlagClr)	否	输入参数有变化： N32H47x_48x 系列不支持 SDIO_FLAG_CEATAF
清除 SDIO 中断标志	void SDIO_ClrIntPendingBit(uint32_t SDIO_IT)	-	否	N32H47x_48x 系列合并到 SDIO_ClrFlag
发送命令并获取 R1 短响应	-	uint32_t SDIO_CmdShortResponse1(uint8_t Cmd, uint32_t Arg, uint32_t timeout)	否	N32H47x_48x 系列新增函数
发送命令并获取 R3 短响应	-	uint32_t SDIO_CmdShortResponse3(uint8_t Cmd, uint32_t Arg)	否	N32H47x_48x 系列新增函数
发送命令并获取 R2 长响应	-	uint32_t SDIO_CmdLongResponse2(uint8_t Cmd, uint32_t Arg)	否	N32H47x_48x 系列新增函数
发送 CMD16 并获取响应	-	uint32_t SDIO_CmdBlockLength(uint32_t BlockSize)	否	N32H47x_48x 系列新增函数
发送 CMD23 并获取响应	-	uint32_t SDIO_CmdBlockCount(uint32_t BlockCnt)	否	N32H47x_48x 系列新增函数
发送 CMD17 并获取响应	-	uint32_t SDIO_CmdReadSingleBlock(uint32_t ReadAdd)	否	N32H47x_48x 系列新增函数
发送 CMD18 并获取响应	-	uint32_t SDIO_CmdReadMultiBlock(uint32_t ReadAdd)	否	N32H47x_48x 系列新增函数
发送 CMD24 并获取响应	-	uint32_t SDIO_CmdWriteSingleBlock(uint32_t WriteAdd)	否	N32H47x_48x 系列新增函数
发送 CMD25 并获取响应	-	uint32_t SDIO_CmdWriteMultiBlock(uint32_t WriteAdd)	否	N32H47x_48x 系列新增函数
发送 CMD32 并获取响应	-	uint32_t SDIO_CmdSDEraseStartAdd(uint32_t StartAdd)	否	N32H47x_48x 系列新增函数
发送 CMD33 并获取响应	-	uint32_t SDIO_CmdSDEraseEndAdd(uint32_t EndAdd)	否	N32H47x_48x 系列新增函数
发送 CMD35 并获取响应	-	uint32_t SDIO_CmdEraseStartAdd(uint32_t StartAdd)	否	N32H47x_48x 系列新增函数

发送 CMD36 并获取响应	-	uint32_t SDIO_CmdEraseEndAdd(uint32_t EndAdd)	否	N32H47x_48x 系列新增函数
发送 CMD38 并获取响应	-	uint32_t SDIO_CmdErase(void)	否	N32H47x_48x 系列新增函数
发送 CMD42 并获取响应	-	uint32_t SDIO_CmdLockUnlock(uint32_t arg)	否	N32H47x_48x 系列新增函数
发送 CMD12 并获取响应	-	uint32_t SDIO_CmdStopTransfer(void)	否	N32H47x_48x 系列新增函数
发送 CMD7 并获取响应	-	uint32_t SDIO_CmdSelDesel(uint16_t RCA)	否	N32H47x_48x 系列新增函数
发送 CMD0 并获取响应	-	uint32_t SDIO_CmdGoIdleState(void)	否	N32H47x_48x 系列新增函数
发送 CMD8 并获取响应	-	uint32_t SDIO_CmdOperCond(void)	否	N32H47x_48x 系列新增函数
发送 CMD55 并获取响应	-	uint32_t SDIO_CmdAppCommand(uint32_t Arg)	否	N32H47x_48x 系列新增函数
发送 ACMD41 并获取响应	-	uint32_t SDIO_CmdAppOperCommand(uint32_t Arg)	否	N32H47x_48x 系列新增函数
发送 ACMD6 并获取响应	-	uint32_t SDIO_CmdBusWidth(uint32_t BusWidth)	否	N32H47x_48x 系列新增函数
发送 ACMD51 并获取响应	-	uint32_t SDIO_CmdSendSCR(void)	否	N32H47x_48x 系列新增函数
发送 CMD2 并获取响应	-	uint32_t SDIO_CmdAllSendCID(void)	否	N32H47x_48x 系列新增函数
发送 CMD10 并获取响应	-	uint32_t SDIO_CmdSendCID(uint16_t RCA)	否	N32H47x_48x 系列新增函数
发送 CMD9 并获取响应	-	uint32_t SDIO_CmdSendCSD(uint16_t RCA)	否	N32H47x_48x 系列新增函数
发送 CMD3 给 SD 卡获取响应和 RCA	-	uint32_t SDIO_CmdSetRelAdd(uint16_t *pRCA)	否	N32H47x_48x 系列新增函数
发送 CMD3 给 MMC 卡并获取响应	-	uint32_t SDIO_CmdSetRelAddMmc(uint16_t RCA)	否	N32H47x_48x 系列新增函数
发送 CMD13 并获取响应	-	uint32_t SDIO_CmdSendStatus(uint32_t Arg)	否	N32H47x_48x 系列新增函数
发送 ACMD13 并获取响应	-	uint32_t SDIO_CmdStatusRegister(void)	否	N32H47x_48x 系列新增函数
发送 CMD1 并获取响应	-	uint32_t SDIO_CmdOpCondition(uint32_t Arg)	否	N32H47x_48x 系列新增函数

发送 CMD6 并获取响应	-	uint32_t SDIO_CmdSwitch(uint32_t Arg)	否	N32H47x_48x 系列新增函数
发送 CMD8 并获取响应	-	uint32_t SDIO_CmdSendEXTCSD(uint32_t Arg)	否	N32H47x_48x 系列新增函数
获取 R1 响应错误状态	-	uint32_t SDIO_GetCmdResp1(uint8_t Cmd, uint32_t Timeout)	否	N32H47x_48x 系列新增函数
获取 R2 响应错误状态	-	uint32_t SDIO_GetCmdResp2(void)	否	N32H47x_48x 系列新增函数
获取 R3 响应错误状态	-	uint32_t SDIO_GetCmdResp3(void)	否	N32H47x_48x 系列新增函数
获取 R6 响应错误状态	-	uint32_t SDIO_GetCmdResp6(uint8_t Cmd, uint16_t *pRCA)	否	N32H47x_48x 系列新增函数
获取 R7 响应错误状态	-	uint32_t SDIO_GetCmdResp7(void)	否	N32H47x_48x 系列新增函数
获取 CMD0 发送错误状态	-	static uint32_t SDIO_GetCmdError(void)	否	N32H47x_48x 系列新增函数

4.3.14 FDCAN

N32H47x_48x系列FDCAN模块符合ISO 11898-1:2015标准，支持CAN 2.0A/B与CAN FD协议，并兼容非ISO标准的Bosch协议。对应N32G45x系列的CAN模块，只支持CAN 2.0A/B。
两者差异较大，无法直接对比，更不能直接使用替换的方式修改。使用时请参考用户手册中的FDCAN模块部分，以及SDK中的FDCAN例程。
另外，FDCAN端口支持管脚全映射，详见用户手册中的GPIO模块中的复用功能描述。

4.3.15 IWDG

IWDG模块主要差异如下：

1. N32G45x 不支持冻结，N32H47x_48x 支持冻结；
2. IWDG 时钟源为 LSI，N32G45x LSI 时钟频率为 40KHz，最小复位时间为 0.1ms，N32H47x_48x LSI 时钟频率为 32KHz，最小复位时间为 0.125ms，所以需要重新计算超时值；

下表是N32G45x系列和N32H47x_48x系列在IWDG模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用IWDG模块驱动API函数可以依照下表做替换，超时时间需要重新换算，详情请参考用户手册：

API 描述	API 名称		差异说明

	N32G45x 系列	N32H47x_48x 系列	API 是否一致	
设置 IWDG_PREDIV 和 IWDG_RELV 寄存器写保护	void IWDG_WriteConfig(uint16_t IWDG_WriteAccess)	void IWDG_WriteConfig(IWDOG_WRITE_CONFIG IWDG_WriteAccess)	是	无
设置驱动 IWDG 的时钟预分频因子	void IWDG_SetPrescalerDiv(uint8_t IWDG_Prescaler)	void IWDG_SetPrescalerDiv(uint8_t IWDG_Prescaler)	是	无
设置 IWDG 重装载值	void IWDG_CntReload(uint16_t Reload)	void IWDG_CntReload(uint16_t Reload)	是	无
将重装载寄存器中的值装载都计数器	void IWDG_ReloadKey(void)	void IWDG_ReloadKey(void)	是	无
使能 IWDG	void IWDG_Enable(void)	void IWDG_Enable(void)	是	无
IWDG 冻结使能/禁能	-	void IWDG_Freeze_Enable(FunctionalState Cmd)	否	N32G45x 系列无此 API
IWDG 状态获取	FlagStatus IWDG_GetStatus(uint16_t IWDG_FLAG)	FlagStatus IWDG_GetStatus(IWDG_STATUS_FLAG IWDG_FLAG)	是	无

4.3.16 WWDG

WWDG模块主要差异如下：

1. N32G45x 窗口值为 7bit，N32H47x_48x 窗口值为 14bit；
2. WWDG 时钟源为 APB1/4096，N32G45x APB1 时钟最大 36MHz，最小复位时间为 0.113ms，N32H47x_48x 当主频为 240MHz 时，APB1 时钟频率为 120MHz，最小复位时间为 0.0341ms，所以需要重新计算窗口值；

下表是N32G45x系列和N32H47x_48x系列在WWDG模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用WWDG模块驱动API函数可以依照下表做替换，窗口时间需要重新换算，详情请参考用户手册：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
复位 WWDG	void WWDG_DeInit(void)	void WWDG_DeInit(void)	是	无
设置 WWDG 驱动时钟的预分频因子	void WWDG_SetPrescalerDiv(uint32_t WWDG_Prescaler)	void WWDG_SetPrescalerDiv(uint32_t WWDG_Prescaler)	是	无
设置窗口值	void WWDG_SetWValue(uint8_t WindowValue)	void WWDG_SetWValue(uint16_t WindowValue)	否	N32G45x 输入值为 7bit，N32G47x_48x 输入值为 14bit
WWDG 提前唤醒中断使能	void WWDG_EnableInt(void)	void WWDG_EnableInt(void)	是	无

设置计数器值	void WWDG_SetCnt(uint8_t Counter)	void WWDG_SetCnt(uint16_t Counter)	否	N32G45x 输入值为 7bit， N32G47x_48x 输入值为 14bit
使能 WWDG	void WWDG_Enable(uint8_t Counter)	void WWDG_Enable(uint16_t Counter)	否	N32G45x 输入值为 7bit， N32G47x_48x 输入值为 14bit
获取 WWDG 提前唤醒中断标志	FlagStatus WWDG_GetEWINTF(void)	FlagStatus WWDG_GetEWINTF(void)	是	无
清除 WWDG 提前唤醒中断标志	void WWDG_ClrEWINTF(void)	void WWDG_ClrEWINTF(void)	是	无

4.3.17FLASH

下表是N32G45x系列和N32H47x_48x系列在FLASH模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用FLASH模块驱动API函数可以依照下表做替换：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
Latency 设置	void FLASH_SetLatency(uint32_t FLASH_Latency)	void FLASH_SetLatency(uint32_t FLASH_Latency)	是	新增参数 FLASH_LATENCY_5
Latency 获取	-	uint8_t FLASH_GetLatency(void)	否	新增函数
预取 buffer 设置	void FLASH_PrefetchBufSet(uint32_t FLASH_PrefetchBuf)	void FLASH_PrefetchBufSet(uint32_t FLASH_PrefetchBuf)	是	无
Icache 复位	void FLASH_iCacheRST(void)	void FLASH_iCacheRST(void)	是	无
Icache 设置	void FLASH_iCacheCmd(uint32_t FLASH_iCache)	void FLASH_iCacheCmd(uint32_t FLASH_iCache)	是	无
FLASH 编程模式设置	void FLASH_SetSMPSELStatus(uint32_t FLASH_smpsel)	-	否	无此函数
FLASH 解锁	void FLASH_Unlock(void)	void FLASH_Unlock(void)	是	无
FLASH 上锁	void FLASH_Lock(void)	void FLASH_Lock(void)	是	无
FLASH 锁状态	-	FlagStatus Flash_GetLockStatus(void)	否	新增函数
选项字节解锁	-	void Option_Bytes_Unlock(void)	否	新增函数
选项字节上锁	-	void Option_Bytes_Lock(void)	否	新增函数
选项字节锁状态	-	FlagStatus OB_GetLockStatus(void)	否	新增函数
FLASH 页擦	FLASH_STS FLASH_EraseOnePage(uint32_t Page_Address)	FLASH_STS FLASH_EraseOnePage(uint32_t Page_Address)	是	页大小：2KB→8KB
FLASH 全擦	FLASH_STS FLASH_MassErase(void)	FLASH_STS FLASH_MassErase(void)	是	无
FLASH 编程	FLASH_STS FLASH_ProgramWord(uint32_t Address, uint32_t Data)	FLASH_STS FLASH_ProgramdoubleWord(uint32_t address, uint32_t data0,uint32_t data1)	否	单字编程改为双字编程，需要同时输入两个 Word 数据
FLASH ROW 编程设置	-	void FLASH_RowProgramSet(FunctionalState Cmd)	否	新增函数
FLASH ROW 区域编程设置	-	void FLASH_RowProgramAreaSet(FunctionalState Cmd)	否	新增函数
FLASH ROW 编程	-	FLASH_STS FLASH_RowProgram(uint32_t address, uint32_t row_num, uint32_t *data)	否	新增函数
选项字节擦除	FLASH_STS FLASH_EraseOB(void)	FLASH_STS FLASH_EraseOB(void)	是	无
选项字节区编程	FLASH_STS FLASH_ProgramOBData(uint32_t Address, uint32_t Data)	FLASH_STS FLASH_ProgramOB_RUDD(uint32_t option_byte_rpd1, uint32_t option_byte_iwdg, uint32_t option_byte_stop,\	否	

		uint32_t option_byte_stdby, uint32_t option_byte_iwdg_stop, uint32_t option_byte_iwdg_stdby,\ uint32_t option_byte_iwdg_sleep, uint32_t option_byte_data0, uint32_t option_byte_data1)		选项字节区编程差异性较大，建议参考 Demo 理解使用
	FLASH_STS FLASH_EnWriteProtection(uint32_t FLASH_Pages)	FLASH_STS FLASH_EnWriteProtection(uint32_t FLASH_Pages)	是	
	FLASH_STS FLASH_ConfigUserOB(uint16_t OB_IWDG, uint16_t OB_STOP, uint16_t OB_STDBY)	FLASH_STS FLASH_ProgramOB_RU2U3(uint32_t option_byte_rpd2, uint32_t option_byte2_nBOOT0, uint32_t option_byte2_nBOOT1,\ uint32_t option_byte2_nSWBOOT0, uint32_t option_byte2_FlashBoot, \ uint32_t option_byte2_BOR, uint32_t option_byte3_NRST)	否	
	-	FLASH_STS FLASH_ProgramOB_CCMSRAM(uint32_t option_byte_CCMSRAM)	否	
FLASH L1 读保护设置	FLASH_STS FLASH_ReadOutProtectionL1(FunctionalState Cmd)	FLASH_STS FLASH_ReadOutProtectionL1(FunctionalState Cmd)	是	无
FLASH L2 读保护使能	FLASH_STS FLASH_ReadOutProtectionL2_ENABLE(void)	FLASH_STS FLASH_ReadOutProtectionL2_ENABLE(void)	是	无
获取选项字节区 USER1 状态	uint32_t FLASH_GetUserOB(void)	uint32_t FLASH_GetUserOB(void)	是	无
获取选项字节区 USER2 状态	-	FlagStatus FLASH_GetUser2(uint32_t option_byte_bit)	否	新增函数
获取选项字节区 DATA0 状态	-	uint32_t FLASH_GetOptionBytes_Data0(void)	否	新增函数
获取选项字节区 DATA1 状态	-	uint32_t FLASH_GetOptionBytes_Data1(void)	否	新增函数
获取 FLASH 写保护状态	uint32_t FLASH_GetWriteProtectionSTS(void)	uint32_t FLASH_GetWriteProtectionSTS(void)	是	无
获取读保护 L1 状态	FlagStatus FLASH_GetReadOutProtectionSTS(void)	FlagStatus FLASH_GetReadOutProtectionSTS(void)	是	无
获取读保护 L2 状态	FlagStatus FLASH_GetReadOutProtectionL2STS(void)	FlagStatus FLASH_GetReadOutProtectionL2STS(void)	是	无
获取预取 buffer 状态	FlagStatus FLASH_GetPrefetchBufSTS(void)	FlagStatus FLASH_GetPrefetchBufSTS(void)	是	无
获取 FLASH 编程模式	FLASH_SMPSEL FLASH_GetSMPSELStatus(void)	-	否	无此函数
FLASH 中断配置	void FLASH_INTConfig(uint32_t FLASH_INT, FunctionalState Cmd)	void FLASH_INTConfig(uint32_t FLASH_INT, FunctionalState Cmd)	是	FLASH_IT_ERROR→FLASH_INT_ERR 新增参数：

				FLASH_INT_ECC1 FLASH_INT_JS FLASH_INT_ECC2 FLASH_INT_DECC FLASH_INT_RPADD
获取 FLASH 具体标志状态	FlagStatus FLASH_GetFlagSTS(uint32_t FLASH_FLAG)	FlagStatus FLASH_GetFlagSTS(uint32_t FLASH_FLAG)	是	新增参数： FLASH_FLAG_ECC1ERR FLASH_FLAG_RDKEYERR FLASH_FLAG_RDXKEYERR FLASH_FLAG_NRDXKEYEN FLASH_FLAG_JSERR FLASH_FLAG_RTPKEYERR FLASH_FLAG_ECC2ERR FLASH_FLAG_DECCRDF FLASH_FLAG_DECCERR FLASH_FLAG_FWORDF FLASH_FLAG_RPADDERR
清除 FLASH 状态	void FLASH_ClearFlag(uint32_t FLASH_FLAG)	void FLASH_ClearFlag(uint32_t FLASH_FLAG)	是	新增参数： FLASH_FLAG_ECC1ERR FLASH_FLAG_JSERR FLASH_FLAG_RTPKEYERR FLASH_FLAG_ECC2ERR FLASH_FLAG_DECCRDF FLASH_FLAG_DECCERR FLASH_FLAG_RPADDERR
获取 FLASH 状态	FLASH_STS FLASH_GetSTS(void)	FLASH_STS FLASH_GetSTS(void)	是	无
等待 FLASH 操作完成	FLASH_STS FLASH_WaitForLastOpt(uint32_t Timeout)	FLASH_STS FLASH_WaitForLastOpt(uint32_t Timeout)	是	新增参数： RowProgramTimeout
CCM 写保护设置	-	void CCM_EnWriteProtection(uint32_t CCM_Pages)	否	新增函数
获取 CCM 写保护状态	-	uint32_t CCM_GetWriteProtectionSTS(void)	否	新增函数
CCM 擦除解锁	-	void CCM_Earse_Unlock(void)	否	新增函数
CCM 擦除使能	-	void CCM_EarseEN(void)	否	新增函数
获取 CCM 擦除状态	-	FlagStatus CCM_EarseSTS(void)	否	新增函数
CCM 模式配置	-	void CCM_ModeSet(FunctionalState Cmd)	否	新增函数
XSPI 解密地址范围设置	-	void XSPI_DESRangeSet(uint32_t start_add,uint32_t end_add)	否	新增函数

FEMC 解密地址范围设置	-	void FEMC_DESRangeSet(uint32_t start_add,uint32_t end_add)	否	新增函数
XPSI/FEMC 解密 KEY 设置	-	void RTP_DESKeySet(uint32_t* DES_key)	否	新增函数
获取解密 KEY 写数量	-	uint32_t GetRTP_DESKeyWnum(void)	否	新增函数
JTAG 封口设置	-	void Jtag_SealSet(FunctionalState Cmd)	否	新增函数
XPSI/FEMC 解密设置	-	void XSPI_FEMC_DESSet(FunctionalState Cmd)	否	新增函数
获取 XSPI/FEMC 配置用户	-	uint32_t Get_XFUID(void)	否	新增函数
获取 CCM 配置用户	-	uint32_t Get_CCMUID(void)	否	新增函数

下图是N32H47x_48x系列在FLASH模块的“OptionByte_config”示例工程代码，对选项字节编程前需要先执行FLASH解锁、选项字节解锁，然后擦除选项字节，最后依次调用选项字节编程函数对选项字节区域进行编程：

```c
int main(void)
{
    FLASH_STS state_value;

    /* USART Init */
    log_init();

    log_info("\nOption Byte configure Test start!\r\n");
    /* Unlocks the FLASH Program Erase Controller */
    FLASH_Unlock();
    /* Unlocks the Option Byte Program Erase Controller */
    Option_Bytes_Unlock();
    /*  Erase Option Byte page */
    state_value = FLASH_EraseOB();

    if(state_value == FLASH_EOP)
    {
        /*  Start Option Byte program */

        /* Disable read protection L1 */
        state_value = FLASH_ProgramOB_RUDD(FLASH_OB_RDP1_DISABLE,FLASH_OB_IWDG_SOFTWARE,FLASH_OB_STOP_NORST,\
                                           FLASH_OB_STDBY_NORST,FLASH_OB_IWDG_STOP_NOFRZ,FLASH_OB_IWDG_STDBY_NOFRZ,\
                                           FLASH_OB_IWDG_SLEEP_NOFRZ,0x55,0xAA);
        if(state_value == FLASH_EOP)
        {
            state_value = FLASH_EnWriteProtection(FLASH_WRP_Pages32to33);
        }
        else
        {
            /* no process*/
        }

        if(state_value == FLASH_EOP)
        {
            state_value = FLASH_ProgramOB_RU2U3(FLASH_OB_RDP2_DISABLE,FLASH_OB2_NBOOT0_SET,FLASH_OB2_NBOOT1_SET,\
                                                FLASH_OB2_NSWBOOT0_SET,FLASH_OB2_FLASHBOOT_SET,0x00,0x00);
        }
        else
        {
            /* no process*/
        }

        if(state_value == FLASH_EOP)
        {
            state_value = FLASH_ProgramOB_CCMSRAM(CCMSRAM_RST_NERASE);
        }
        else
        {
            /* no process*/
        }
        if(state_value == FLASH_EOP)
        {
            log_info("Option Byte configure OK!\r\n");
        }
        else
        {
            log_info("Option Byte configure failed!\r\n");
        }
    }
    else
    {
        log_info("Option Byte erase failed!\r\n");
    }

    while (1)
    {
    }
}
```

4.3.18MISC

下表是N32G45x系列和N32H47x_48x系列在misc模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用misc模块驱动API函数可以依照下表做替换：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
NVIC 优先级分组	void NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup)	void NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup)	是	无
中断优先级配置	void NVIC_Init(NVIC_InitType* NVIC_InitStruct)	void NVIC_Init(NVIC_InitType* NVIC_InitStruct)	是	无
中断向量表位置配置	void NVIC_SetVectorTable(uint32_t NVIC_VectTab, uint32_t Offset)	void NVIC_SetVectorTable(uint32_t NVIC_VectTab, uint32_t Offset)	是	无
低功耗模式配置	void NVIC_SystemLPConfig(uint8_t LowPowerMode, FunctionalState Cmd)	void NVIC_SystemLPConfig(uint8_t LowPowerMode, FunctionalState Cmd)	是	无
Systick 时钟源配置	void SysTick_CLKSourceConfig(uint32_t SysTick_CLKSource)	void SysTick_CLKSourceConfig(uint32_t SysTick_CLKSource)	是	无

N32G45x系列与N32H47x_48x系列在misc模块函数功能一致，可以直接替换使用。

4.3.19USART

下表是N32G45x系列和N32H47x_48x系列在USART模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用USART模块驱动API函数可以依照下表做替换；其中USARTx参数差异性说明统一为UART4→USART4，新增UART8。

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
USART 复位	void USART_DeInit(USART_Module* USARTx)	void USART_DeInit(USART_Module* USARTx)	是	无
USART 初始化	void USART_Init(USART_Module* USARTx, USART_InitType* USART_InitStruct)	void USART_Init(USART_Module* USARTx, USART_InitType* USART_InitStruct)	是	BaudRate 最大值由 1.5M⟶15M 新增 OverSampling 配置参数
USART 结构体初始化	void USART_StructInit(USART_InitType* USART_InitStruct)	void USART_StructInit(USART_InitType* USART_InitStruct)	是	新增 OverSampling 配置参数
USART 时钟初始化	void USART_ClockInit(USART_Module* USARTx, USART_ClockInitType* USART_ClockInitStruct)	void USART_ClockInit(USART_Module* USARTx, USART_ClockInitType* USART_ClockInitStruct)	是	无
USART 时钟结构体初始化	void USART_ClockStructInit(USART_ClockInitType* USART_ClockInitStruct)	void USART_ClockStructInit(USART_ClockInitType* USART_ClockInitStruct)	是	无

USART 模块使能	void USART_Enable(USART_Module* USARTx, FunctionalState Cmd)	void USART_Enable(USART_Module* USARTx, FunctionalState Cmd)	是	无
USART 中断配置	void USART_ConfigInt(USART_Module* USARTx, uint32_t USART_INT, FunctionalState Cmd)	void USART_ConfigInt(USART_Module* USARTx, uint32_t USART_INT, FunctionalState Cmd)	是	新增参数： USART_INT_RTOE USART_INT_TXFTE USART_INT_RXFTE USART_INT_RXFEE USART_INT_TXFEE USART_INT_RXFFE USART_INT_TXFFE
USART DMA 配置	void USART_EnableDMA(USART_Module* USARTx, uint32_t USART_DMAReq, FunctionalState Cmd)	void USART_EnableDMA(USART_Module* USARTx, uint32_t USART_DMAReq, FunctionalState Cmd)	是	无
USART 节点地址设置	void USART_SetAddr(USART_Module* USARTx, uint8_t USART_Addr)	void USART_SetAddr(USART_Module* USARTx, uint8_t USART_Addr)	是	无
USART 唤醒模式设置	void USART_ConfigWakeUpMode(USART_Module* USARTx, uint32_t USART_WakeUpMode)	void USART_ConfigWakeUpMode(USART_Module* USARTx, uint32_t USART_WakeUpMode)	是	无
USART 静默模式唤醒设置	void USART_EnableRcvWakeUp(USART_Module* USARTx, FunctionalState Cmd)	void USART_EnableRcvWakeUp(USART_Module* USARTx, FunctionalState Cmd)	是	无
USART LIN 空闲帧长度设置	void USART_ConfigLINBreakDetectLength(USART_Module* USARTx, uint32_t USART_LINBreakDetectLength)	void USART_ConfigLINBreakDetectLength(USART_Module* USARTx, uint32_t USART_LINBreakDetectLength)	是	无
LIN 模式设置	void USART_EnableLIN(USART_Module* USARTx, FunctionalState Cmd)	void USART_EnableLIN(USART_Module* USARTx, FunctionalState Cmd)	是	无
发送数据	void USART_SendData(USART_Module* USARTx, uint32_t Data)	void USART_SendData(USART_Module* USARTx, uint32_t Data)	是	无
接收数据	uint32_t USART_ReceiveData(USART_Module* USARTx)	uint32_t USART_ReceiveData(USART_Module* USARTx)	是	无
发送断开帧	void USART_SendBreak(USART_Module* USARTx)	void USART_SendBreak(USART_Module* USARTx)	是	无
保护时间设置	void USART_SetGuardTime(USART_Module* USARTx, uint8_t USART_GuardTime)	void USART_SetGuardTime(USART_Module* USARTx, uint8_t USART_GuardTime)	是	无
系统时钟分频设置	void USART_SetPrescaler(USART_Module* USARTx, uint8_t USART_Prescaler)	void USART_SetPrescaler(USART_Module* USARTx, uint8_t USART_Prescaler)	是	无
智能卡模式设置	void USART_EnableSmartCard(USART_Module* USARTx, FunctionalState Cmd)	void USART_EnableSmartCard(USART_Module* USARTx, FunctionalState Cmd)	是	无
智能卡 NACK 设置	void USART_SetSmartCardNACK(USART_Module* USARTx, FunctionalState Cmd)	void USART_SetSmartCardNACK(USART_Module* USARTx, FunctionalState Cmd)	是	无
USART 半双工模式设置	void USART_EnableHalfDuplex(USART_Module* USARTx, FunctionalState Cmd)	void USART_EnableHalfDuplex(USART_Module* USARTx, FunctionalState Cmd)	是	无

红外功耗模式设置	void USART_ConfigIrDAMode(USART_Module* USARTx, uint32_t USART_IrDAMode)	void USART_ConfigIrDAMode(USART_Module* USARTx, uint32_t USART_IrDAMode)	是	无
红外模式设置	void USART_EnableIrDA(USART_Module* USARTx, FunctionalState Cmd)	void USART_EnableIrDA(USART_Module* USARTx, FunctionalState Cmd)	是	无
获取 USART 状态	FlagStatus USART_GetFlagStatus(USART_Module* USARTx, uint32_t USART_FLAG)	FlagStatus USART_GetFlagStatus(USART_Module* USARTx, uint32_t USART_FLAG)	是	新增参数： USART_FLAG_FELOSE USART_FLAG_NELOSE USART_FLAG_PELOSE USART_FLAG_RTO USART_FLAG_TXFT USART_FLAG_RXFT USART_FLAG_RXFE USART_FLAG_TXFE USART_FLAG_RXFF USART_FLAG_TXFF
清除 USART 状态位	void USART_ClrFlag(USART_Module* USARTx, uint32_t USART_FLAG)	void USART_ClrFlag(USART_Module* USARTx, uint32_t USART_FLAG)	是	新增参数： USART_FLAG_FELOSE USART_FLAG_NELOSE USART_FLAG_PELOSE
清除 USART RTO 状态位	-	void USART_ClrRTOFlag(USART_Module* USARTx)	否	新增函数
获取中断状态	INTStatus USART_GetIntStatus(USART_Module* USARTx, uint32_t USART_INT)	INTStatus USART_GetIntStatus(USART_Module* USARTx, uint32_t USART_INT)	是	函数含义发生变化 N32G45x 此函数用于判断中断是否使能且对应标志位是否置起 N32H47x_48x 此函数仅用于判断中断是否使能，需和 USART_GetFlagStatus 函数一起使用和达到 N32G45x 一致效果 新增参数： USART_INT_RTOE USART_INT_TXFTE USART_INT_RXFTE USART_INT_RXFEE USART_INT_TXFEE USART_INT_RXFFE USART_INT_TXFFE
清除 USART 状态位	void USART_ClrIntPendingBit(USART_Module* USARTx, uint16_t USART_INT)	-	否	无此函数，此函数功能和 USART_ClrFlag 功能一致

USART 空闲帧设置	-	void USART_IdleFrameSet(USART_Module* USARTx,FunctionalState Cmd)	否	新增函数
USART 引脚互换功能	-	void USART_PinSwapSet(USART_Module* USARTx,FunctionalState Cmd)	否	新增函数
USART 驱动器使能起始时间配置	-	void USART_CfgDriverAssertTime(USART_Module* USARTx,uint32_t Time)	否	新增函数
USART 驱动器使能禁止时间配置	-	void USART_CfgDriverdeassertTime(USART_Module* USARTx,uint32_t Time)	否	新增函数
USART 驱动器使能极性设置	-	void USART_DriverPolaritySet(USART_Module* USARTx,FunctionalState Cmd)	否	新增函数
USART 驱动器使能模式设置	-	void USART_DriverModeSet(USART_Module* USARTx,FunctionalState Cmd)	否	新增函数
USART FEF 数据丢弃设置	-	void USART_FEFDiscardSet(USART_Module* USARTx,FunctionalState Cmd)	否	新增函数
USART NEF 数据丢弃设置	-	void USART_NEFDiscardSet(USART_Module* USARTx,FunctionalState Cmd)	否	新增函数
USART PEF 数据丢弃设置	-	void USART_PEFDiscardSet(USART_Module* USARTx,FunctionalState Cmd)	否	新增函数
USART 接收超时设置	-	void USART_RTOSet(USART_Module* USARTx,FunctionalState Cmd)	否	新增函数
获取 USART TxFIFO 有效数据个数	-	uint32_t USART_GetTxFIFO_Num(USART_Module* USARTx)	否	新增函数
获取 USART RxFIFO 有效数据个数	-	uint32_t USART_GetRxFIFO_Num(USART_Module* USARTx)	否	新增函数
USART RxFIFO 阈值配置	-	void USART_CfgRxFIFOThreshold(USART_Module* USARTx,uint32_t threshold)	否	新增函数
USART TxFIFO 阈值配置	-	void USART_CfgTxFIFOThreshold(USART_Module* USARTx,uint32_t threshold)	否	新增函数
USART FIFO 清 0	-	void USART_ClrFIFO(USART_Module* USARTx)	否	新增函数
USART FIFO 模式设置	-	void USART_FIFOModeSet(USART_Module* USARTx,FunctionalState Cmd)	否	新增函数
USART 空闲帧宽度设置		void USART_IdleFrameWidthSet(USART_Module* USARTx,uint32_t Width)	否	新增函数
USART 接收超时配置	-	void USART_CfgRTOWidth(USART_Module* USARTx,uint32_t Width)	否	新增函数

下图是N32G45x系列（左）和N32H47x_48x系列（右）在USART模块的“Interrupt”示例工程代码对比：

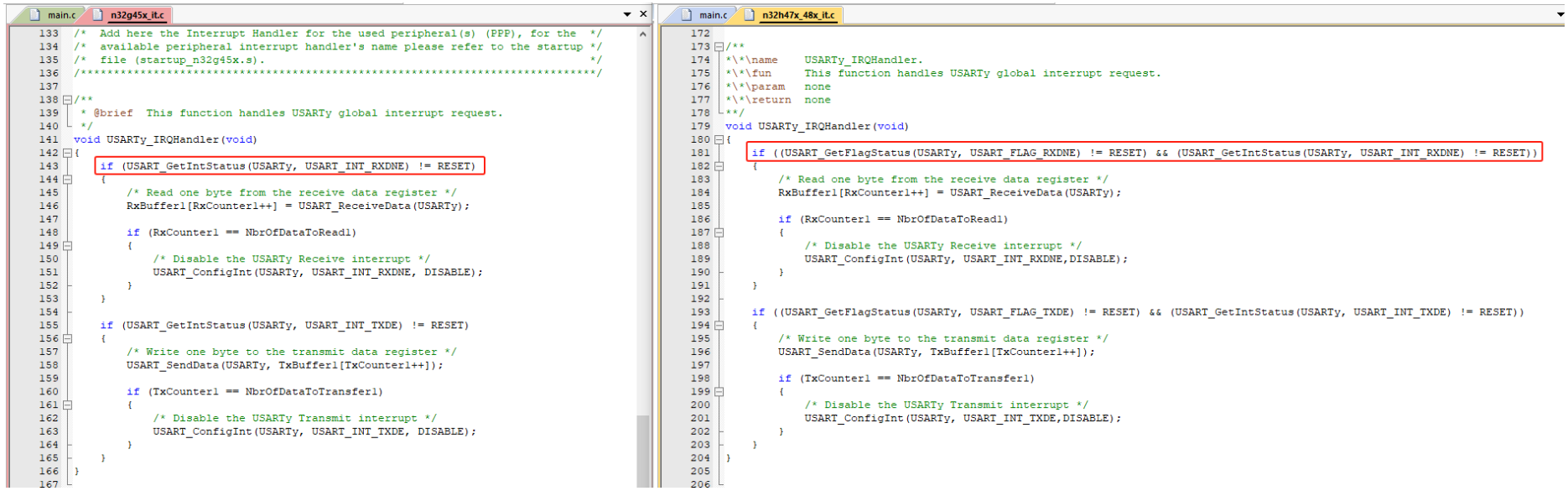
main.c | n32g45x_it.c

```
82  */
83  int main(void)
84  {
85      /* System Clocks Configuration */
86      RCC_Configuration();
87
88      /* NVIC configuration */
89      NVIC_Configuration();
90
91      /* Configure the GPIO ports */
92      GPIO_Configuration();
93
94      /* USARTy and USARTz configuration ------------------------------------------------------*/
95      USART_InitStructure.BaudRate            = 115200;
96      USART_InitStructure.WordLength          = USART_WL_8B;
97      USART_InitStructure.StopBits            = USART_STPB_1;
98      USART_InitStructure.Parity              = USART_PE_NO;
99      USART_InitStructure.HardwareFlowControl = USART_HFCTRL_NONE;
100     USART_InitStructure.Mode                = USART_MODE_RX | USART_MODE_TX;
101
102     /* Configure USARTy and USARTz */
103     USART_Init(USARTy, &USART_InitStructure);
104     USART_Init(USARTz, &USART_InitStructure);
105
106     /* Enable USARTy Receive and Transmit interrupts */
107     USART_ConfigInt(USARTy, USART_INT_RXDNE, ENABLE);
108     USART_ConfigInt(USARTy, USART_INT_TXDE, ENABLE);
109
110     /* Enable USARTz Receive and Transmit interrupts */
111     USART_ConfigInt(USARTz, USART_INT_RXDNE, ENABLE);
112     USART_ConfigInt(USARTz, USART_INT_TXDE, ENABLE);
113
114     /* Enable the USARTy and USARTz */
115     USART_Enable(USARTy, ENABLE);
116     USART_Enable(USARTz, ENABLE);
117
118     /* Wait until end of transmission from USARTy to USARTz */
119     while (RxCounter2 < RxBufferSize2)
120     {
121     }
122
123     /* Wait until end of transmission from USARTz to USARTy */
124     while (RxCounter1 < RxBufferSize1)
125     {
126     }
```

main.c | n32h47x_48x_it.c

```
85  int main(void)
86  {
87      /* System Clocks Configuration */
88      RCC_Configuration();
89
90      /* NVIC configuration */
91      NVIC_Configuration();
92
93      /* Configure the GPIO ports */
94      GPIO_Configuration();
95
96      /* USARTy and USARTz configuration */
97      USART_InitStructure.BaudRate            = 115200;
98      USART_InitStructure.WordLength          = USART_WL_8B;
99      USART_InitStructure.StopBits            = USART_STPB_1;
100     USART_InitStructure.Parity              = USART_PE_NO;
101     USART_InitStructure.HardwareFlowControl = USART_HFCTRL_NONE;
102     USART_InitStructure.OverSampling        = USART_16OVER;
103     USART_InitStructure.Mode                = USART_MODE_RX | USART_MODE_TX;
104
105     /* Configure USARTy and USARTz */
106     USART_Init(USARTy, &USART_InitStructure);
107     USART_Init(USARTz, &USART_InitStructure);
108
109     /* Enable USARTy Receive and Transmit interrupts */
110     USART_ConfigInt(USARTy, USART_INT_RXDNE,ENABLE);
111     USART_ConfigInt(USARTy, USART_INT_TXDE,ENABLE);
112
113     /* Enable USARTz Receive and Transmit interrupts */
114     USART_ConfigInt(USARTz, USART_INT_RXDNE,ENABLE);
115     USART_ConfigInt(USARTz, USART_INT_TXDE,ENABLE);
116
117     /* Enable the USARTy and USARTz */
118     USART_Enable(USARTy, ENABLE);
119     USART_Enable(USARTz, ENABLE);
120
121     /* Wait until end of transmission from USARTy to USARTz */
122     while (RxCounter2 < RxBufferSize2)
123     {
124     }
125
126     /* Wait until end of transmission from USARTz to USARTy */
127     while (RxCounter1 < RxBufferSize1)
128     {
129     }
```

过采样配置



```c
/*  Add here the Interrupt Handler for the used peripheral(s) (PPP), for the  */
/*  available peripheral interrupt handler's name please refer to the startup */
/*  file (startup_n32g45x.s).                                                 */
/*****************************************************************************/

/**
 * @brief  This function handles USARTy global interrupt request.
 */
void USARTy_IRQHandler(void)
{
    if (USART_GetIntStatus(USARTy, USART_INT_RXDNE) != RESET)
    {
        /* Read one byte from the receive data register */
        RxBuffer1[RxCounter1++] = USART_ReceiveData(USARTy);

        if (RxCounter1 == NbrOfDataToRead1)
        {
            /* Disable the USARTy Receive interrupt */
            USART_ConfigInt(USARTy, USART_INT_RXDNE, DISABLE);
        }
    }

    if (USART_GetIntStatus(USARTy, USART_INT_TXDE) != RESET)
    {
        /* Write one byte to the transmit data register */
        USART_SendData(USARTy, TxBuffer1[TxCounter1++]);

        if (TxCounter1 == NbrOfDataToTransfer1)
        {
            /* Disable the USARTy Transmit interrupt */
            USART_ConfigInt(USARTy, USART_INT_TXDE, DISABLE);
        }
    }
}
```

```c
/**
 *\*\name    USARTy_IRQHandler.
 *\*\fun     This function handles USARTy global interrupt request.
 *\*\param   none
 *\*\return  none
 **/
void USARTy_IRQHandler(void)
{
    if ((USART_GetFlagStatus(USARTy, USART_FLAG_RXDNE) != RESET) && (USART_GetIntStatus(USARTy, USART_INT_RXDNE) != RESET))
    {
        /* Read one byte from the receive data register */
        RxBuffer1[RxCounter1++] = USART_ReceiveData(USARTy);

        if (RxCounter1 == NbrOfDataToRead1)
        {
            /* Disable the USARTy Receive interrupt */
            USART_ConfigInt(USARTy, USART_INT_RXDNE,DISABLE);
        }
    }

    if ((USART_GetFlagStatus(USARTy, USART_FLAG_TXDE) != RESET) && (USART_GetIntStatus(USARTy, USART_INT_TXDE) != RESET))
    {
        /* Write one byte to the transmit data register */
        USART_SendData(USARTy, TxBuffer1[TxCounter1++]);

        if (TxCounter1 == NbrOfDataToTransfer1)
        {
            /* Disable the USARTy Transmit interrupt */
            USART_ConfigInt(USARTy, USART_INT_TXDE,DISABLE);
        }
    }
}
```

USART 配置流程基本一致，区别在于 N32H47x_48x 系列配置增加了过采样配置；在中断判断中 USART_GetIntStatus 函数由 USART_GetFlagStatus 和 USART_GetIntStatus 代替。

4.3.20**DBG**

DBG模块主要差异如下：

1. N32H47x_48x 支持 SHRTIM 调试暂停；

下表是N32G45x系列和N32H47x_48x系列在DBG模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用DBG模块驱动API函数可以依照下表做替换，详情请参考用户手册：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
DBG 外设配置	void DBG_ConfigPeriph(uint32_t DBG_Periph, FunctionalState Cmd)	void DBG_ConfigPeriph(uint32_t DBG_Periph, FunctionalState Cmd)	否	函数命名及输入参数 DBG_Periph 部分不一致： N32G45x 系列参数： DBG_SLEEP DBG_STOP DBG_STDBY

				DBG_IWDG_STOP DBG_WWDG_STOP DBG_TIM1_STOP DBG_TIM2_STOP DBG_TIM3_STOP DBG_TIM4_STOP DBG_CAN1_STOP DBG_I2C1SMBUS DBG_I2C2SMBUS DBG_TIM8_STOP DBG_TIM5_STOP DBG_TIM6_STOP DBG_TIM7_STOP DBG_CAN2_STOP N32H47x_48x 系列参数： DBG_SLEEP DBG_STOP DBG_STANDBY DBG_IWDG_STOP DBG_WWDG_STOP DBG_I2C1SMBUS_TIMEOUT DBG_I2C2SMBUS_TIMEOUT DBG_I2C3SMBUS_TIMEOUT DBG_I2C4SMBUS_TIMEOUT DBG_ATIM1_STOP DBG_ATIM2_STOP DBG_ATIM3_STOP DBG_BTIM1_STOP DBG_BTIM2_STOP DBG_GTIM1_STOP DBG_GTIM2_STOP DBG_GTIM3_STOP DBG_GTIM4_STOP DBG_GTIM5_STOP DBG_GTIM6_STOP DBG_GTIM7_STOP DBG_GTIM8_STOP DBG_GTIM9_STOP DBG_GTIM10_STOP DBG_SHRTIM1_STOP
获取 UCID	void GetUCID(uint8_t *UCIDbuf)	void GetUCID(uint8_t *UCIDbuf)	是	无
获取 UID	void GetUID(uint8_t *UIDbuf)	void GetUID(uint8_t *UIDbuf)	是	无
获取 MCU ID	void GetDBGMCU_ID(uint8_t *DBGMCU_IDbuf)	void GetDBGMCU_ID(uint8_t *DBGMCU_IDbuf)	是	无

获取芯片版本	uint32_t DBG_GetRevNum(void)	uint32_t DBG_GetRevNum(void)	是	无
获取芯片型号	uint32_t DBG_GetDevNum(void)	uint32_t DBG_GetDevNum(void)	是	无
获取 FLASH 大小	uint32_t DBG_GetFlashSize(void)	uint32_t DBG_GetFlashSize(void)	是	无
获取 SRAM 大小	uint32_t DBG_GetSramSize(void)	uint32_t DBG_GetSramSize(void)	是	无

4.3.21 RTC

RTC模块主要差异如下：

1. N32G45x 无参考时钟输入，N32H47x_48x 支持参考时钟输入；
2. N32G45x 将 BKP 单独作为一个模块，N32H47x_48x 将备份寄存器和 RTC 合成了一个模块；
3. N32G45x BKP 和 RTC 模块是分开的，函数和驱动也是分开的，N32H47x_48x BKP 和 RTC 模块合并在一起，函数和驱动合并在一起，所以和 BKP 相关函数，N32G45x 和 N32H47x_48x 函数名和传参均不一样；
4. N32G45x 支持一个入侵引脚，N32H47x_48x 支持三个入侵引脚；
5. N32G45x 支持 40 个 16bit 备份寄存器，N32H47x_48x 支持 20 个 32bit 备份寄存器；
6. N32G45x 入侵中断没有连接到 EXTI 线，N32H47x_48x RTC 入侵中断连接 EXTI19；

下表是N32G430系列和N32G43x系列在RTC模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用RTC模块驱动API函数可以依照下表做替换：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
RTC 复位	ErrorStatus RTC_DeInit(void)	ErrorStatus RTC_Deinitializes(void)	是	
RTC 初始化	ErrorStatus RTC_Init(RTC_InitType* RTC_InitStruct)	ErrorStatus RTC_Init(RTC_InitType* RTC_InitStruct)	是	
RTC 结构体初始化	void RTC_StructInit(RTC_InitType* RTC_InitStruct)	void RTC_StructInit(RTC_InitType* RTC_InitStruct)	是	
使能写保护	void RTC_EnableWriteProtection(FunctionalState Cmd)	void RTC_EnableWriteProtection(FunctionalState Cmd)	是	
进入初始化模式	ErrorStatus RTC_EnterInitMode(void)	ErrorStatus RTC_EnterInitMode(void)	是	
退出初始化模式	void RTC_ExitInitMode(void)	void RTC_ExitInitMode(void)	是	
等待同步	ErrorStatus RTC_WaitForSynchro(void)	ErrorStatus RTC_WaitForSynchro(void)	是	
使能参考时钟	-	ErrorStatus RTC_EnableRefClock(FunctionalState Cmd)	否	N32G45x 系列无此 API
使能影子寄存器旁路	void RTC_EnableBypassShadow(FunctionalState Cmd)	void RTC_EnableBypassShadow(FunctionalState Cmd)	是	
设置时间	ErrorStatus RTC_ConfigTime(uint32_t RTC_Format, RTC_TimeType* RTC_TimeStruct)	ErrorStatus RTC_ConfigTime(uint32_t RTC_Format, RTC_TimeType* RTC_TimeStruct)	是	

时间结构体初始化	void RTC_TimeStructInit(RTC_TimeType* RTC_TimeStruct)s	void RTC_TimeStructInit(RTC_TimeType* RTC_TimeStruct)	是	
获取时间	void RTC_GetTime(uint32_t RTC_Format, RTC_TimeType* RTC_TimeStruct)	void RTC_GetTime(uint32_t RTC_Format, RTC_TimeType* RTC_TimeStruct)	是	
获取亚秒值	uint32_t RTC_GetSubSecond(void)	uint32_t RTC_GetSubSecond(void)	是	
设置日期	ErrorStatus RTC_SetDate(uint32_t RTC_Format, RTC_DateType* RTC_DateStruct)	ErrorStatus RTC_SetDate(uint32_t RTC_Format, RTC_DateType* RTC_DateStruct)	是	
日期结构体初始化	void RTC_DateStructInit(RTC_DateType* RTC_DateStruct)	void RTC_DateStructInit(RTC_DateType* RTC_DateStruct)	是	
获取日期	void RTC_GetDate(uint32_t RTC_Format, RTC_DateType* RTC_DateStruct)	void RTC_GetDate(uint32_t RTC_Format, RTC_DateType* RTC_DateStruct)	是	
日期结构体初始化	void RTC_DateStructInit(RTC_DateType* RTC_DateStruct)	void RTC_Date_Struct_Initializes(RTC_DateType* RTC_DateStruct)	是	
设置闹钟	void RTC_SetAlarm(uint32_t RTC_Format, uint32_t RTC_Alarm, RTC_AlarmType* RTC_AlarmStruct)	void RTC_SetAlarm(uint32_t RTC_Format, uint32_t RTC_Alarm, RTC_AlarmType* RTC_AlarmStruct)	是	
闹钟结构体初始化	void RTC_AlarmStructInit(RTC_AlarmType* RTC_AlarmStruct)	void RTC_AlarmStructInit(RTC_AlarmType* RTC_AlarmStruct)	是	
获取闹钟	void RTC_GetAlarm(uint32_t RTC_Format, uint32_t RTC_Alarm, RTC_AlarmType* RTC_AlarmStruct)	void RTC_GetAlarm(uint32_t RTC_Format, uint32_t RTC_Alarm, RTC_AlarmType* RTC_AlarmStruct)	是	
使能闹钟	ErrorStatus RTC_EnableAlarm(uint32_t RTC_Alarm, FunctionalState Cmd)	ErrorStatus RTC_EnableAlarm(uint32_t RTC_Alarm, FunctionalState Cmd)	是	
配置闹钟亚秒	void RTC_ConfigAlarmSubSecond(uint32_t RTC_Alarm, uint32_t RTC_AlarmSubSecondValue, uint32_t RTC_AlarmSubSecondMask)	void RTC_ConfigAlarmSubSecond(uint32_t RTC_Alarm, uint32_t RTC_AlarmSubSecondValue, uint32_t RTC_AlarmSubSecondMask)	是	
获取闹钟亚秒	uint32_t RTC_GetAlarmSubSecond(uint32_t RTC_Alarm)	uint32_t RTC_GetAlarmSubSecond(uint32_t RTC_Alarm)	是	
配置唤醒时钟源	void RTC_ConfigWakeUpClock(uint32_t RTC_WakeUpClock)	void RTC_ConfigWakeUpClock(uint32_t RTC_WakeUpClock)	是	
设置唤醒计数器重装载值	void RTC_SetWakeUpCounter(uint32_t RTC_WakeUpCounter)	void RTC_SetWakeUpCounter(uint32_t RTC_WakeUpCounter)	是	
获取唤醒计数器重装载值	uint32_t RTC_GetWakeUpCounter(void)	uint32_t RTC_GetWakeUpCounter(void)	是	
使能唤醒	ErrorStatus RTC_EnableWakeUp(FunctionalState Cmd)	ErrorStatus RTC_EnableWakeUp(FunctionalState Cmd)	是	
夏时令配置	void RTC_ConfigDayLightSaving(uint32_t RTC_DayLightSaving, uint32_t RTC_StoreOperation)	void RTC_ConfigDayLightSaving(uint32_t RTC_DayLightSaving, uint32_t RTC_StoreOperation)	是	
获取夏时令操作记录	uint32_t RTC_GetStoreOperation(void)	uint32_t RTC_GetStoreOperation(void)	是	
输出配置	void RTC_ConfigOutput(uint32_t RTC_Output, uint32_t RTC_OutputPolarity)	void RTC_ConfigOutput(uint32_t RTC_Output, uint32_t RTC_OutputPolarity)	是	
输入 2 配置	-	void RTC_EnableOutput2(FunctionalState Cmd)	否	N32G45x 系列无此 API

使能 Tamper 输出	-	void RTC_EnableTampOutput(FunctionalState Cmd)	否	N32G45x 系列无此 API
使能校准输出	void RTC_EnableCalibOutput(FunctionalState Cmd)	void RTC_EnableCalibOutput(FunctionalState Cmd)	是	
配置校准输出	void RTC_ConfigCalibOutput(uint32_t RTC_CalibOutput)	void RTC_ConfigCalibOutput(uint32_t RTC_CalibOutput)	是	
精密校准配置	ErrorStatus RTC_ConfigSmoothCalib(uint32_t RTC_SmoothCalibPeriod, uint32_t RTC_SmoothCalibPlusPulses, uint32_t RTC_SmouthCalibMinusPulsesValue)	ErrorStatus RTC_ConfigSmoothCalib(uint32_t RTC_SmoothCalibPeriod, uint32_t RTC_SmoothCalibPlusPulses, uint32_t RTC_SmouthCalibMinusPulsesValue)	是	
使能时间戳	void RTC_EnableTimeStamp(uint32_t RTC_TimeStampEdge, FunctionalState Cmd)	void RTC_EnableTimeStamp(uint32_t RTC_TimeStampEdge, FunctionalState Cmd)	是	
获取时间戳	void RTC_GetTimeStamp(uint32_t RTC_Format, RTC_TimeType* RTC_StampTimeStruct, RTC_DateType* RTC_StampDateStruct)	void RTC_GetTimeStamp(uint32_t RTC_Format, RTC_TimeType* RTC_StampTimeStruct, RTC_DateType* RTC_StampDateStruct)	是	
获取时间戳亚秒	uint32_t RTC_GetTimeStampSubSecond(void)	uint32_t RTC_GetTimeStampSubSecond(void)	是	
输出类型配置	void RTC_ConfigOutputType(uint32_t RTC_OutputType)	void RTC_ConfigOutputType(uint32_t RTC_OutputType)	是	
平移配置	ErrorStatus RTC_ConfigSynchroShift(uint32_t RTC_ShiftAddFS, uint32_t RTC_ShiftSub1s)	ErrorStatus RTC_ConfigSynchroShift(uint32_t RTC_ShiftAdd1S, uint32_t RTC_ShiftSubFS)	否	N32G45x 为减一秒，加偏移亚秒，N32H47x_48x 为加一秒，减偏移亚秒
中断配置	void RTC_ConfigInt(uint32_t RTC_INT, FunctionalState Cmd)	void RTC_ConfigInt(uint32_t RTC_INT, FunctionalState Cmd)	否	N32H47x_48x 比 N32G45x 多了一个时间戳中断
获取状态标志位	FlagStatus RTC_GetFlagStatus(uint32_t RTC_FLAG)	FlagStatus RTC_GetFlagStatus(uint32_t RTC_FLAG)	否	N32H47x_48x 比 N32G45x 多了入侵标志位
清除标志位	void RTC_ClrFlag(uint32_t RTC_FLAG)	void RTC_ClrFlag(uint32_t RTC_FLAG)	否	N32H47x_48x 比 N32G45x 多了入侵标志位
获取中断状态标志位	INTStatus RTC_GetITStatus(uint32_t RTC_INT)	INTStatus RTC_GetITStatus(uint32_t RTC_INT)	否	N32H47x_48x 比 N32G45x 多了时间戳和入侵标志位
清除中断标志位	void RTC_ClrIntPendingBit(uint32_t RTC_INT)	void RTC_ClrIntPendingBit(uint32_t RTC_INT)	否	N32H47x_48x 比 N32G45x 多了时间戳和入侵标志位
使能唤醒 TSC	RTC_EnableWakeUpTsc	-	否	N32H47x_48x 无此 API
BCD 码转成 2 进制	static uint8_t RTC_Bcd2ToByte(uint8_t Value)	static uint8_t RTC_Bcd2ToByte(uint8_t Value)	是	
2 进制转成 BCD 码	static uint8_t RTC_ByteToBcd2(uint8_t Value)	static uint8_t RTC_ByteToBcd2(uint8_t Value)	是	
N32G45x BKP 和 RTC 模块是分开的，驱动也是分开的，N32H47x_48x BKP 和 RTC 模块合并在一起，驱动也合并在了一起，所以和 BKP 相关函数，N32G45x 和 N32H47x_48x 函数名和传参均不一样				
入侵触发方式配置	-	void RTC_TamperTriggerConfig(uint32_t RTC_Tamper, uint32_t RTC_TamperTrigger)	否	N32G45x 系列无此 API

入侵使能	void BKP_TPEnable(FunctionalState Cmd)	void RTC_TamperCmd(uint32_t RTC_Tamper, FunctionalState NewState)	否	N32G45x 只有一个引脚支持入侵检测；N32H47x_48x 有 3 个引脚支持入侵检测
配置入侵过滤器计数	-	void RTC_TamperFilterConfig(uint32_t RTC_TamperFilter)	否	N32G45x 系列无此 API
配置入侵采样频率	-	void RTC_TamperSamplingFreqConfig(uint32_t RTC_TamperSamplingFreq)	否	N32G45x 系列无此 API
配置入侵引脚预充电	-	void RTC_TamperPinsPrechargeDuration(uint32_t RTC_TamperPrechargeDuration)	否	N32G45x 系列无此 API
时间戳触发入侵检测事件	-	void RTC_TimeStampOnTamperDetectionCmd(FunctionalState NewState)	否	N32G45x 系列无此 API
入侵上拉使能	-	void RTC_TamperPullUpCmd(FunctionalState NewState)	否	N32G45x 系列无此 API
入侵擦除备份寄存器配置	void BKP_ConfigTPLevel(uint16_t BKP_TamperPinLevel)	void RTC_EnableTampErase(uint32_t RTC_Tamper_Erase, FunctionalState NewState)	否	
入侵激活时间戳使能	-	void RTC_TamperTAMPTSCmd(FunctionalState NewState)	否	N32G45x 系列无此 API
入侵中断使能	void BKP_TPIntEnable(FunctionalState Cmd)	void RTC_TamperIECmd(uint32_t TAMPxIE, FunctionalState NewState)	否	N32G45x 支持一个入侵引脚，N32H47x_48x 支持三个入侵引脚
获取入侵标志位	FlagStatus BKP_GetTEFlag(void)	FlagStatus RTC_GetFlagStatus(uint32_t RTC_FLAG)	否	
清除入侵标志位	void BKP_ClrTEFlag(void)	void RTC_ClrFlag(uint32_t RTC_FLAG)	否	
获取入侵中断标志位	INTStatus BKP_GetTINTFlag(void)	INTStatus RTC_GetITStatus(uint32_t RTC_INT)	否	
清除入侵中断标志位	void BKP_ClrTINTFlag(void)	void RTC_ClrIntPendingBit(uint32_t RTC_INT)	否	
写备份寄存器	void BKP_WriteBkpData(uint16_t BKP_DAT, uint16_t Data)	void RTC_BKUPRgWrite(uint8_t register_num, uint32_t Data)	否	N32G45x 有 42 个 16bit 备份寄存器；N32H47x_48x 有 20 个 32bit 备份寄存器
读备份寄存器	uint16_t BKP_ReadBkpData(uint16_t BKP_DAT)	uint32_t RTC_BKUPRgRead(uint8_t register_num)	否	N32G45x 有 42 个 16bit 备份寄存器；N32H47x_48x 有 20 个 32bit 备份寄存器

如下为N32G45x和N32H47x_48x的日历配置流程对比，读BKP的API不一样。

```c
int main(void)
{
    /*!< At this stage the microcontroller clock setting is already configured,
         this is done through SystemInit() function which is called from startup
         file (startup_n32g45x_xx.s) before to branch to application main.
         To reconfigure the default setting of SystemInit() function, refer to
         system_n32g45x.c file
       */
    /* Initialize LEDs on n32g45x-EVAL board */
    LEDInit(LED1_PORT, LED1_PIN);
    LEDOff(LED1_PORT, LED1_PIN);
    /* Initialize USART,TX: PC10 RX: PC11*/
    log_init();
    log_info("RTC Init");
    /* RTC date time alarm default value*/
    RTC_DateAndTimeDefaultVale();
    /* Enable the PWR clock */
    RCC_EnableAPB1PeriphClk(RCC_APB1_PERIPH_PWR | RCC_APB1_PERIPH_BKP, ENABLE);

    RCC_ClrFlag();
    /* Allow access to RTC */
    PWR_BackupAccessEnable(ENABLE);
    if (USER_WRITE_BKP_DAT1_DATA != BKP_ReadBkpData(BKP_DAT1) )
    {
       /* Backup data register value is not correct or not yet programmed (when
          the first time the program is executed) */
       log_info("\r\n\n RTC not yet configured....");
       /* RTC clock source select */
       RTC_CLKSourceConfig(RTC_CLK_SRC_TYPE_LSE, true, true);
       RTC_PrescalerConfig();
       log_info("\r\n RTC configured....");
       /* Adjust time by values entered by the user on the hyperterminal */
       RTC_DateRegulate();
       RTC_TimeRegulate();
       BKP_WriteBkpData(BKP_DAT1, USER_WRITE_BKP_DAT1_DATA);
    }
    /* Configure the PA11 pin to generate an EXTI interrupt
       in which the calendar value is printed (externally feed
       the 1HZ signal output on PC13 to PA11 to produce a 1s EXTI interrupt)*/
    EXTI_PA7_Configuration();
    EXTI_ClrITPendBit(EXTI_LINE7);
    /* Calibrate output 1Hz signal */
    RTC_ConfigCalibOutput(RTC_CALIB_OUTPUT_1HZ);
    /* Calibrate output config,push pull */
    RTC_ConfigOutputType(RTC_OUTPUT_PUSHPULL);
    /* Calibrate output enable*/
    RTC_EnableCalibOutput(ENABLE);
    log_info("\r\n RTC Config end....");
    while (1);
}
```

```c
*\*\fun     Main program.
*\*\
**/
int main(void)
{
    /*!< At this stage the microcontroller clock setting is already configured,
         this is done through SystemInit() function which is called from startup
         file (startup_n32h47x_48x_xx.s) before to branch to application main.
         To reconfigure the default setting of SystemInit() function, refer to
         system_n32h47x_48x.c file
       */
    /* Initialize USART */
    log_init();
    log_info("\r\n RTC Init \r\n");
    /* RTC date time alarm default value*/
    RTC_DateAndTimeDefaultVale();
    /* Enable the PWR clock */
    RCC_EnableAPB1PeriphClk(RCC_APB1_PERIPH_PWR, ENABLE);
    /* Allow access to RTC */
    PWR_BackupAccessEnable(ENABLE);
    if (USER_WRITE_BKP_DAT1_DATA != RTC_BKUPRgRead(1))
    {
       /* RTC clock source select */
       if(SUCCESS==RTC_CLKSourceConfig(RTC_CLK_SRC_TYPE_LSE, true, false))
       {
          RTC_PrescalerConfig();
          log_info("\r\n RTC configured....");
          /* Adjust time by values entered by the user on the hyperterminal */
          RTC_DateRegulate();
          RTC_TimeRegulate();
          RTC_BKUPRgWrite(1, USER_WRITE_BKP_DAT1_DATA);
          log_info("\r\n RTC Init Success\r\n");
       }
       else
       {
          log_info("\r\n RTC Init Faile\r\n");
       }
    }
    /* Configure the PA11 pin to generate an EXTI interrupt
       in which the calendar value is printed (externally feed
       the 1HZ signal output on PC13 to PA11 to produce a 1s EXTI interrupt)*/
    EXTI_ClrITPendBit(EXTI_LINE7);
    EXTI_PA7_Configuration();
    /* Calibrate output 1Hz signal */
    RTC_ConfigCalibOutput(RTC_CALIB_OUTPUT_1HZ);
    /* Calibrate output config,push pull */
    RTC_ConfigOutputType(RTC_OUTPUT_PUSHPULL);
    /* Calibrate output enable*/
    RTC_EnableCalibOutput(ENABLE);
    log_info("\r\n RTC Config end....");
    while (1);
}
```

4.3.22 USBFSD

USBFSD模块主要差异如下：

1. 修改模块命名，在 N32G45x 中叫 USB，在 N32H47x_48x 中叫做 USBFSD；
2. 在 N32G45x 中 USB 驱动命名为 usb_xx.c/usb_xx.h，在 N32H47x_48x 中 USBFSD 驱动命名为 usbfsd_xx.c/usbfsd_xx.h，可直接替换；
3. N32G45x 不支持无晶体模式，N32H47x_48x 支持无晶体模式，N32H47x_48x 无晶体模式可参考 HID_Keyboard_Xtal_Less Demo；
4. N32G45x USBFS 和 CAN1 共用 512byte SRAM，N32H47x_48x USBFSD 单独用 512byte SRAM，所以 N32H47x_48 USBFSD 和 CAN 可以同时使用；
5. 当使用 HSE_PLL 作为 USB 时钟源时，N32G45x 系统时钟频率只能是 144MHz、96MHz、72MHz 和 48MHz，N32H47x_48x 系统时钟可以是 240MHz、192MHz、144MHz、96MHz、72MHz 和 48MHz；

4.3.23 USBHS

N32G45x不支持USBHS模块，N32H47x_48x支持USBHS，USBHS支持双角色 （主机和设备），USBHS使用可参考Demo；

4.3.24 FEMC

N32G45x不支持FEMC模块，N32H47x_48x支持FEMC模块，FEMC使用可参考Demo；

4.3.25 SHRTIM

N32G45x不支持SHRTIM模块，N32H47x_48x支持SHRTIM模块，SHRTIM使用可参考Demo及SHRTIM的相关文档

4.3.26 DVP

DVP模块N32G45x系列和N32H47x_48x系列设计上差异较大，建议用户直接参考N32H47x_48x系列例程。

4.3.27 TIM

下表是N32G45x系列和N32H47x_48x系列在TIM模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用PWR模块驱动API函数可以依照下表做替换，互联关系相关的请参考用户手册使用：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
TIM 复位	void TIM_DeInit(TIM_Module* TIMx)	void TIM_DeInit(TIM_Module* TIMx)	否	TIM 编号不一致： N32H47x_48x 的 ATIM1-3 是高级定时器， N32G45x 的 TIM1/TIM8 是高级定时器； N32H47x_48x 的 GTIM1-7 是通用定时器， N32G45x 的 TIM2-5 是通用定时器； N32H47x_48x 的 BTIM1-2 是基础定时器， N32G45x 的 TIM6/TIM7 是基础定时器； N32H47x_48x 的 GTIM8-10 是新增通用定时器。
TIM 基本配置，配置定时器的分频，预装载值，计数方式，重复计数器，输入信号源	void TIM_InitTimeBase(TIM_Module* TIMx, TIM_TimeBaseInitType* TIM_TimeBaseInitStruct)	void TIM_InitTimeBase(TIM_Module* TIMx, TIM_TimeBaseInitType* TIM_TimeBaseInitStruct)	否	TIM 编号不一致。 输入信号不一致： N32H47x_48x 的互联关系增加很多，需要查询用户手册互联关系使用。
TIM 通道 1 配置，配置输出模式，输出使能，互补通道输出使能，比较值，输出极性，互补输出极性，输出空闲状态，互补输出空闲状态	void TIM_InitOc1(TIM_Module* TIMx, OCInitType* TIM_OCInitStruct)	void TIM_InitOc1(TIM_Module* TIMx, OCInitType* TIM_OCInitStruct)	否	TIM 编号不一致。 输出模式不一致： N32H47x_48x 增加如下模式： TIM_OCMODE_OPMOD_RETRIG1 TIM_OCMODE_OPMOD_RETRIG2 TIM_OCMODE_COMBI_PWM1 TIM_OCMODE_COMBI_PWM2
TIM 通道 2 配置，配置输出模式，输出使能，互补通道输出使能，比较值，输出极性，互补输出极性，输出空闲状态，互补输出空闲状态	void TIM_InitOc2(TIM_Module* TIMx, OCInitType* TIM_OCInitStruct)	void TIM_InitOc2(TIM_Module* TIMx, OCInitType* TIM_OCInitStruct)	否	TIM 编号不一致。 输出模式不一致： N32H47x_48x 增加如下模式： TIM_OCMODE_OPMOD_RETRIG1 TIM_OCMODE_OPMOD_RETRIG2 TIM_OCMODE_COMBI_PWM1 TIM_OCMODE_COMBI_PWM2

TIM 通道 3 配置，配置输出模式，输出使能，互补通道输出使能，比较值，输出极性，互补输出极性，输出空闲状态，互补输出空闲状态	void TIM_InitOc3(TIM_Module* TIMx, OCInitType* TIM_OCInitStruct)	void TIM_InitOc3(TIM_Module* TIMx, OCInitType* TIM_OCInitStruct)	否	TIM 编号不一致。 输出模式不一致： N32H47x_48x 增加如下模式： TIM_OCMODE_OPMOD_RETRIG1 TIM_OCMODE_OPMOD_RETRIG2 TIM_OCMODE_COMBI_PWM1 TIM_OCMODE_COMBI_PWM2
TIM 通道 4 配置，配置输出模式，输出使能，互补通道输出使能，比较值，输出极性，互补输出极性，输出空闲状态，互补输出空闲状态	void TIM_InitOc4(TIM_Module* TIMx, OCInitType* TIM_OCInitStruct)	void TIM_InitOc4(TIM_Module* TIMx, OCInitType* TIM_OCInitStruct)	否	TIM 编号不一致。 输出模式不一致： N32H47x_48x 增加如下模式： TIM_OCMODE_OPMOD_RETRIG1 TIM_OCMODE_OPMOD_RETRIG2 TIM_OCMODE_COMBI_PWM1 TIM_OCMODE_COMBI_PWM2
TIM 通道 5 配置，配置输出模式，输出使能，互补通道输出使能，比较值，输出极性，互补输出极性，输出空闲状态，互补输出空闲状态	void TIM_InitOc5(TIM_Module* TIMx, OCInitType* TIM_OCInitStruct)	void TIM_InitOc5(TIM_Module* TIMx, OCInitType* TIM_OCInitStruct)	否	TIM 编号不一致。
TIM 通道 6 配置，配置输出模式，输出使能，互补通道输出使能，比较值，输出极性，互补输出极性，输出空闲状态，互补输出空闲状态	void TIM_InitOc6(TIM_Module* TIMx, OCInitType* TIM_OCInitStruct)	void TIM_InitOc6(TIM_Module* TIMx, OCInitType* TIM_OCInitStruct)	否	TIM 编号不一致。
TIM 输入捕获配置	void TIM_ICInit(TIM_Module* TIMx, TIM_ICInitType* TIM_ICInitStruct)	void TIM_ICInit(TIM_Module* TIMx, TIM_ICInitType* TIM_ICInitStruct)	否	TIM 编号不一致。
TIM PWM 捕获配置	void TIM_ConfigPwmIc(TIM_Module* TIMx, TIM_ICInitType* TIM_ICInitStruct)	void TIM_ConfigPwmIc(TIM_Module* TIMx, TIM_ICInitType* TIM_ICInitStruct)	否	TIM 编号不一致。

TIM 刹车配置	void TIM_ConfigBkdt(TIM_Module* TIMx, TIM_BDTRInitType* TIM_BDTRInitStruct)	void TIM_ConfigBkdt(TIM_Module* TIMx, TIM_BDTRInitType* TIM_BDTRInitStruct)	否	TIM 编号不一致。 N32H47x_48x 刹车配置增加一路刹车 2，刹车 1 增加双向刹车配置（部分刹车源有单独的极性控制，不在本函数中配置），N32H47x_48x 刹车源配置需单独配置。具体功能差异可参考用户手册和 DEMO。
刹车 1 滤波配置	-	void TIM_BreakFiltConfig(TIM_Module* TIMx, TIM_FiltInitType* TIM_FiltInitStruct)	否	TIM 编号不一致。 N32H47x_48x 新增刹车 1 滤波配置
刹车 2 滤波配置	-	void TIM_Break2FiltConfig(TIM_Module* TIMx, TIM_FiltInitType* TIM_FiltInitStruct)	否	TIM 编号不一致。 N32H47x_48x 新增刹车 2 滤波配置
刹车 1 滤波使能	-	void TIM_BreakFiltEnable(TIM_Module* TIMx,FunctionalState Cmd)	否	TIM 编号不一致。 N32H47x_48x 新增刹车 1 滤波使能
刹车 2 滤波使能	-	void TIM_Break2FiltEnable(TIM_Module* TIMx,FunctionalState Cmd)	否	TIM 编号不一致。 N32H47x_48x 新增刹车 2 滤波使能
刹车 1 源使能	-	void TIM_BreakInputSourceEnable(TIM_Module* TIMx, uint32_t Source, uint32_t Polarity, FunctionalState Cmd)	否	TIM 编号不一致。 N32H47x_48x 新增刹车源选择极性和单独使能
刹车 2 源使能	-	void TIM_Break2InputSourceEnable(TIM_Module* TIMx, uint32_t Source, uint32_t Polarity, FunctionalState Cmd)	否	TIM 编号不一致。 N32H47x_48x 新增刹车源选择极性和单独使能
双向刹车 1 解除	-	void TIM_BidirectionDisarm(TIM_Module* TIMx)	否	TIM 编号不一致。 N32H47x_48x 新增双向刹车功能
双向刹车 1 使能	-	void TIM_BidirectionRearm(TIM_Module* TIMx)	否	TIM 编号不一致。 N32H47x_48x 新增双向刹车功能
双向刹车 2 解除	-	void TIM_Bidirection2Disarm(TIM_Module* TIMx)	否	TIM 编号不一致。 N32H47x_48x 新增双向刹车功能
双向刹车 2 使能	-	void TIM_Bidirection2Rearm(TIM_Module* TIMx)	否	TIM 编号不一致。 N32H47x_48x 新增双向刹车功能
初始化结构体	void TIM_InitTimBaseStruct(TIM_TimeBaseInitType* TIM_TimeBaseInitStruct)	void TIM_InitTimBaseStruct(TIM_TimeBaseInitType* TIM_TimeBaseInitStruct)	否	结构体初始化输入参数改变
初始化结构体	void TIM_InitOcStruct(OCInitType* TIM_OCInitStruct)	void TIM_InitOcStruct(OCInitType* TIM_OCInitStruct)	是	-
初始化结构体	void TIM_InitIcStruct(TIM_ICInitType* TIM_ICInitStruct)	void TIM_InitIcStruct(TIM_ICInitType* TIM_ICInitStruct)	是	-

初始化结构体	void TIM_InitBkdtStruct(TIM_BDTRInitType* TIM_BDTRInitStruct)	void TIM_InitBkdtStruct(TIM_BDTRInitType* TIM_BDTRInitStruct)	否	结构体初始化输入参数改变
TIM 使能	void TIM_Enable(TIM_Module* TIMx, FunctionalState Cmd)	void TIM_Enable(TIM_Module* TIMx, FunctionalState Cmd)	否	TIM 编号不一致。
TIM 主输出使能	void TIM_EnableCtrlPwmOutputs(TIM_Module* TIMx, FunctionalState Cmd)	void TIM_EnableCtrlPwmOutputs(TIM_Module* TIMx, FunctionalState Cmd)	否	TIM 编号不一致。
TIM 中断使能	void TIM_ConfigInt(TIM_Module* TIMx, uint16_t TIM_IT, FunctionalState Cmd)	void TIM_ConfigInt(TIM_Module* TIMx, uint32_t TIM_IT, FunctionalState Cmd)	否	TIM 编号不一致。 中断输入参数增加： TIM_INT_CC5 TIM_INT_CC6 TIM_INT_CC7 TIM_INT_CC8 TIM_INT_CC9
TIM 生成事件	void TIM_GenerateEvent(TIM_Module* TIMx, uint16_t TIM_EventSource)	void TIM_GenerateEvent(TIM_Module* TIMx, uint32_t TIM_EventSource)	否	TIM 编号不一致。 时间输入参数增加： TIM_EVT_SRC_BREAK2
TIM 的 DMA 配置	void TIM_ConfigDma(TIM_Module* TIMx, uint32_t TIM_DMABase, uint32_t TIM_DMABurstLength)	void TIM_ConfigDma(TIM_Module* TIMx, uint32_t TIM_DMABase, uint32_t TIM_DMABurstLength)	否	TIM 编号不一致。 DMA 初始地址输入参数修改： TIM_DMABASE_CTRL1 TIM_DMABASE_CTRL2 TIM_DMABASE_STS TIM_DMABASE_EVTGEN TIM_DMABASE_SMCTRL TIM_DMABASE_DMAINTEN TIM_DMABASE_CAPCMPMOD1 TIM_DMABASE_CAPCMPMOD2 TIM_DMABASE_CAPCMPMOD3 TIM_DMABASE_CAPCMPEN TIM_DMABASE_CAPCMPDAT1 TIM_DMABASE_CAPCMPDAT2 TIM_DMABASE_CAPCMPDAT3 TIM_DMABASE_CAPCMPDAT4 TIM_DMABASE_CAPCMPDAT5 TIM_DMABASE_CAPCMPDAT6 TIM_DMABASE_PSC TIM_DMABASE_AR TIM_DMABASE_CNT TIM_DMABASE_REPCNT

				TIM_DMABASE_BKDT TIM_DMABASE_CAPCMPDAT7 TIM_DMABASE_CAPCMPDAT8 TIM_DMABASE_CAPCMPDAT9 TIM_DMABASE_BKFR TIM_DMABASE_C1FILT TIM_DMABASE_C2FILT TIM_DMABASE_C3FILT TIM_DMABASE_C4FILT TIM_DMABASE_FILTO TIM_DMABASE_INSEL TIM_DMABASE_AF1 TIM_DMABASE_AF2 TIM_DMABASE_BKFR2 TIM_DMABASE_SLIDFPSC DMA 长度输入参数增加： TIM_DMABURST_LENGTH_1TRANSFER 到 TIM_DMABURST_LENGTH_35TRANSFERS
DMA 请求源配置	void TIM_EnableDma(TIM_Module* TIMx, uint32_t TIM_DMASource, FunctionalState Cmd)	void TIM_EnableDma(TIM_Module* TIMx, uint32_t TIM_DMASource, FunctionalState Cmd)	否	TIM 编号不一致。
TIM 内部时钟模式	void TIM_ConfigInternalClk(TIM_Module* TIMx)	void TIM_ConfigInternalClk(TIM_Module* TIMx)	否	TIM 编号不一致。
TIM 内部信号作为时钟配置	void TIM_ConfigInternalTrigToExt(TIM_Module* TIMx, uint16_t TIM_InputTriggerSource)	void TIM_ConfigInternalTrigToExt(TIM_Module * TIMx, uint32_t TIM_InputTriggerSource)	否	TIM 编号不一致。 内部信号源修改，具体信号源参考用户手册： TIM_TRIG_SEL_IN_TR0 TIM_TRIG_SEL_IN_TR1 TIM_TRIG_SEL_IN_TR2 TIM_TRIG_SEL_IN_TR3 TIM_TRIG_SEL_IN_TR4 TIM_TRIG_SEL_IN_TR5 TIM_TRIG_SEL_IN_TR6 TIM_TRIG_SEL_IN_TR7 TIM_TRIG_SEL_IN_TR8 TIM_TRIG_SEL_IN_TR9 TIM_TRIG_SEL_IN_TR10 TIM_TRIG_SEL_IN_TR11 TIM_TRIG_SEL_IN_TR12 TIM_TRIG_SEL_IN_TR13 TIM_TRIG_SEL_IN_TR14
TIM 外部信号作为时钟配置	void TIM_ConfigExtTrigAsClk(TIM_Module* TIMx, uint16_t TIM_TIxExternalCLKSource, uint16_t IcPolarity, uint16_t ICFilter)	void TIM_ConfigExtTrigAsClk(TIM_Module* TIMx, uint32_t TIM_TIxExternalCLKSource, uint32_t IcPolarity, uint32_t ICFilter)	否	TIM 编号不一致。

外部时钟模式 1 配置	void TIM_ConfigExtClkMode1(TIM_Module* TIMx, uint16_t TIM_ExtTRGPrescaler,uint16_t TIM_ExtTRGPolarity,uint16_t ExtTRGFilter)	void TIM_ConfigExtClkMode1(TIM_Module* TIMx,uint32_t TIM_ETRInputSource,uint32_t TIM_ExtTRGPrescaler,uint32_t TIM_ExtTRGPolarity,uint32_t ExtTRGFilter)	否	TIM 编号不一致。 ETR 信号源参数增加，具体信号源参考用户手册: TIM_CAPETRSEL_0 TIM_CAPETRSEL_1 TIM_CAPETRSEL_2 TIM_CAPETRSEL_3 TIM_CAPETRSEL_4 TIM_CAPETRSEL_5 TIM_CAPETRSEL_6 TIM_CAPETRSEL_7 TIM_CAPETRSEL_8 TIM_CAPETRSEL_9 TIM_CAPETRSEL_10 TIM_CAPETRSEL_11 TIM_CAPETRSEL_12 TIM_CAPETRSEL_13
外部时钟模式 2 配置	void TIM_ConfigExtClkMode2(TIM_Module* TIMx,uint16_t TIM_ExtTRGPrescaler,uint16_t TIM_ExtTRGPolarity,uint16_t ExtTRGFilter)	void TIM_ConfigExtClkMode2(TIM_Module* TIMx,uint32_t TIM_ETRInputSource,uint32_t TIM_ExtTRGPrescaler,uint32_t TIM_ExtTRGPolarity,uint32_t ExtTRGFilter)	否	TIM 编号不一致。 ETR 信号源参数增加，具体信号源参考用户手册: TIM_CAPETRSEL_0 TIM_CAPETRSEL_1 TIM_CAPETRSEL_2 TIM_CAPETRSEL_3 TIM_CAPETRSEL_4 TIM_CAPETRSEL_5 TIM_CAPETRSEL_6 TIM_CAPETRSEL_7 TIM_CAPETRSEL_8 TIM_CAPETRSEL_9 TIM_CAPETRSEL_10 TIM_CAPETRSEL_11 TIM_CAPETRSEL_12 TIM_CAPETRSEL_13
ETR 配置	void TIM_ConfigExtTrig(TIM_Module* TIMx,uint16_t TIM_ExtTRGPrescaler,uint16_t TIM_ExtTRGPolarity,uint16_t ExtTRGFilter)	void TIM_ConfigExtTrig(TIM_Module* TIMx,uint32_t TIM_ExtTRGPrescaler,uint32_t TIM_ExtTRGPolarity,uint32_t ExtTRGFilter)	否	TIM 编号不一致。
TIM 分频配置	void TIM_ConfigPrescaler(TIM_Module* TIMx, uint16_t Prescaler, uint16_t TIM_PSCReloadMode)	void TIM_ConfigPrescaler(TIM_Module* TIMx, uint32_t Prescaler, uint32_t TIM_PSCReloadMode)	否	TIM 编号不一致。

TIM 计数模式配置	void TIM_ConfigCntMode(TIM_Module* TIMx, uint16_t CntMode)	void TIM_ConfigCntMode(TIM_Module* TIMx, uint32_t CntMode)	否	TIM 编号不一致。
TIM 触发输入源配置	void TIM_SelectInputTrig(TIM_Module* TIMx, uint16_t TIM_InputTriggerSource)	void TIM_SelectInputTrig(TIM_Module* TIMx, uint32_t TIM_InputTriggerSource)	否	TIM 编号不一致。 输入源信号参数增加，具体信号源参考用户手册： TIM_TRIG_SEL_IN_TR0 TIM_TRIG_SEL_IN_TR1 TIM_TRIG_SEL_IN_TR2 TIM_TRIG_SEL_IN_TR3 TIM_TRIG_SEL_IN_TR4 TIM_TRIG_SEL_IN_TR5 TIM_TRIG_SEL_IN_TR6 TIM_TRIG_SEL_IN_TR7 TIM_TRIG_SEL_IN_TR8 TIM_TRIG_SEL_IN_TR9 TIM_TRIG_SEL_IN_TR10 TIM_TRIG_SEL_IN_TR11 TIM_TRIG_SEL_IN_TR12 TIM_TRIG_SEL_IN_TR13 TIM_TRIG_SEL_IN_TR14 TIM_TRIG_SEL_TI1F_ED TIM_TRIG_SEL_TI1FP1 TIM_TRIG_SEL_TI2FP2 TIM_TRIG_SEL_ETRF
编码器模式配置	void TIM_ConfigEncoderInterface(TIM_Module* TIMx,uint16_t TIM_EncoderMode,uint16_t TIM_IC1Polarity,uint16_t TIM_IC2Polarity)	void TIM_ConfigEncoderInterface(TIM_Module* TIMx,uint32_t TIM_EncoderMode,uint32_t TIM_IC1Polarity,uint32_t TIM_IC2Polarity)	否	TIM 编号不一致。 编码器模式输入参数增加： TIM_ENCODE_QUA_MODE_SINGLE_TI1 TIM_ENCODE_QUA_MODE_SINGLE_TI2 TIM_ENCODE_DUL_CLKPLUS_MODE1 TIM_ENCODE_DUL_CLKPLUS_MODE2 TIM_ENCODE_SINGLE_CLKPLUS_MODE1 TIM_ENCODE_SINGLE_CLKPLUS_MODE2
通道 1 强制输出	void TIM_ConfigForcedOc1(TIM_Module* TIMx, uint16_t TIM_ForcedAction)	void TIM_ConfigForcedOc1(TIM_Module* TIMx, uint32_t TIM_ForcedAction)	否	TIM 编号不一致。
通道 2 强制输出	void TIM_ConfigForcedOc2(TIM_Module* TIMx, uint16_t TIM_ForcedAction)	void TIM_ConfigForcedOc2(TIM_Module* TIMx, uint16_t TIM_ForcedAction)	否	TIM 编号不一致。
通道 3 强制输出	void TIM_ConfigForcedOc3(TIM_Module* TIMx, uint16_t TIM_ForcedAction)	void TIM_ConfigForcedOc3(TIM_Module* TIMx, uint16_t TIM_ForcedAction)	否	TIM 编号不一致。
通道 4 强制输出	void TIM_ConfigForcedOc4(TIM_Module* TIMx, uint16_t TIM_ForcedAction)	void TIM_ConfigForcedOc4(TIM_Module* TIMx, uint16_t TIM_ForcedAction)	否	TIM 编号不一致。

通道 5 强制输出	void TIM_ConfigForcedOc5(TIM_Module* TIMx, uint16_t TIM_ForcedAction)	void TIM_ConfigForcedOc5(TIM_Module* TIMx, uint16_t TIM_ForcedAction)	否	TIM 编号不一致。
通道 6 强制输出	void TIM_ConfigForcedOc6(TIM_Module* TIMx, uint16_t TIM_ForcedAction)	void TIM_ConfigForcedOc6(TIM_Module* TIMx, uint16_t TIM_ForcedAction)	否	TIM 编号不一致。
AR 预装载使能	void TIM_ConfigArPreload(TIM_Module* TIMx, FunctionalState Cmd)	void TIM_ConfigArPreload(TIM_Module* TIMx, FunctionalState Cmd)	否	TIM 编号不一致。
COM 控制更新选择	void TIM_SelectComEvt(TIM_Module* TIMx, FunctionalState Cmd)	void TIM_SelectComEvt(TIM_Module* TIMx, FunctionalState Cmd)	否	TIM 编号不一致。
DMA 请求使能	void TIM_SelectCapCmpDmaSrc(TIM_Module* TIMx, FunctionalState Cmd)	void TIM_SelectCapCmpDmaSrc(TIM_Module* TIMx, FunctionalState Cmd)	否	TIM 编号不一致。
捕获/比较预装载控制位使能	void TIM_EnableCapCmpPreloadControl(TIM_Module* TIMx, FunctionalState Cmd)	void TIM_EnableCapCmpPreloadControl(TIM_Module* TIMx, FunctionalState Cmd)	否	TIM 编号不一致。
CCDAT1 预装载使能	void TIM_ConfigOc1Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	void TIM_ConfigOc1Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	否	TIM 编号不一致。
CCDAT2 预装载使能	void TIM_ConfigOc2Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	void TIM_ConfigOc2Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	否	TIM 编号不一致。
CCDAT3 预装载使能	void TIM_ConfigOc3Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	void TIM_ConfigOc3Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	否	TIM 编号不一致。
CCDAT4 预装载使能	void TIM_ConfigOc4Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	void TIM_ConfigOc4Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	否	TIM 编号不一致。
CCDAT5 预装载使能	void TIM_ConfigOc5Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	void TIM_ConfigOc5Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	否	TIM 编号不一致。
CCDAT6 预装载使能	void TIM_ConfigOc6Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	void TIM_ConfigOc6Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	否	TIM 编号不一致。
CCDAT7 预装载使能	-	void TIM_ConfigOc7Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	否	新增
CCDAT8 预装载使能	-	void TIM_ConfigOc8Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	否	新增
CCDAT9 预装载使能	-	void TIM_ConfigOc9Preload(TIM_Module* TIMx, uint32_t TIM_OCPreload)	否	新增
通道 1 快速使能	void TIM_ConfigOc1Fast(TIM_Module* TIMx, uint32_t TIM_OCFast)	void TIM_ConfigOc1Fast(TIM_Module* TIMx, uint32_t TIM_OCFast)	否	TIM 编号不一致。
通道 2 快速使能	void TIM_ConfigOc2Fast(TIM_Module* TIMx, uint32_t TIM_OCFast)	void TIM_ConfigOc2Fast(TIM_Module* TIMx, uint32_t TIM_OCFast)	否	TIM 编号不一致。

通道 3 快速使能	void TIM_ConfigOc3Fast(TIM_Module* TIMx, uint32_t TIM_OCFast)	void TIM_ConfigOc3Fast(TIM_Module* TIMx, uint32_t TIM_OCFast)	否	TIM 编号不一致。
通道 4 快速使能	void TIM_ConfigOc4Fast(TIM_Module* TIMx, uint32_t TIM_OCFast)	void TIM_ConfigOc4Fast(TIM_Module* TIMx, uint32_t TIM_OCFast)	否	TIM 编号不一致。
通道 5 快速使能	void TIM_ConfigOc5Fast(TIM_Module* TIMx, uint32_t TIM_OCFast)	void TIM_ConfigOc5Fast(TIM_Module* TIMx, uint32_t TIM_OCFast)	否	TIM 编号不一致。
通道 6 快速使能	void TIM_ConfigOc6Fast(TIM_Module* TIMx, uint32_t TIM_OCFast)	void TIM_ConfigOc6Fast(TIM_Module* TIMx, uint32_t TIM_OCFast)	否	TIM 编号不一致。
通道 1 输出比较清除使能	void TIM_ClrOc1Ref(TIM_Module* TIMx, uint16_t TIM_OCClear)	void TIM_ClrOc1Ref(TIM_Module* TIMx, uint16_t TIM_OCClear)	否	TIM 编号不一致。
通道 2 输出比较清除使能	void TIM_ClrOc2Ref(TIM_Module* TIMx, uint16_t TIM_OCClear)	void TIM_ClrOc2Ref(TIM_Module* TIMx, uint16_t TIM_OCClear)	否	TIM 编号不一致。
通道 3 输出比较清除使能	void TIM_ClrOc3Ref(TIM_Module* TIMx, uint16_t TIM_OCClear)	void TIM_ClrOc3Ref(TIM_Module* TIMx, uint16_t TIM_OCClear)	否	TIM 编号不一致。
通道 4 输出比较清除使能	void TIM_ClrOc4Ref(TIM_Module* TIMx, uint16_t TIM_OCClear)	void TIM_ClrOc4Ref(TIM_Module* TIMx, uint16_t TIM_OCClear)	否	TIM 编号不一致。
通道 5 输出比较清除使能	void TIM_ClrOc5Ref(TIM_Module* TIMx, uint16_t TIM_OCClear)	void TIM_ClrOc5Ref(TIM_Module* TIMx, uint16_t TIM_OCClear)	否	TIM 编号不一致。
通道 6 输出比较清除使能	void TIM_ClrOc6Ref(TIM_Module* TIMx, uint16_t TIM_OCClear)	void TIM_ClrOc6Ref(TIM_Module* TIMx, uint16_t TIM_OCClear)	否	TIM 编号不一致。
通道 x 输出比较清除触发源选择	-	void TIM_ClrOcRefInputSource(TIM_Module* TIMx, uint32_t OCRefClearInputSelect,uint32_t OCRefClearInputSource)	否	新增，用于选择触发源。
通道 1 极性选择	void TIM_ConfigOc1Polarity(TIM_Module* TIMx, uint16_t OcPolarity)	void TIM_ConfigOc1Polarity(TIM_Module* TIMx, uint16_t OcPolarity)	否	TIM 编号不一致。
通道 2 极性选择	void TIM_ConfigOc2Polarity(TIM_Module* TIMx, uint16_t OcPolarity)	void TIM_ConfigOc2Polarity(TIM_Module* TIMx, uint16_t OcPolarity)	否	TIM 编号不一致。
通道 3 极性选择	void TIM_ConfigOc3Polarity(TIM_Module* TIMx, uint16_t OcPolarity)	void TIM_ConfigOc3Polarity(TIM_Module* TIMx, uint16_t OcPolarity)	否	TIM 编号不一致。
通道 4 极性选择	void TIM_ConfigOc4Polarity(TIM_Module* TIMx, uint16_t OcPolarity)	void TIM_ConfigOc4Polarity(TIM_Module* TIMx, uint16_t OcPolarity)	否	TIM 编号不一致。
通道 5 极性选择	void TIM_ConfigOc5Polarity(TIM_Module* TIMx, uint16_t OcPolarity)	void TIM_ConfigOc5Polarity(TIM_Module* TIMx, uint16_t OcPolarity)	否	TIM 编号不一致。
通道 6 极性选择	void TIM_ConfigOc6Polarity(TIM_Module* TIMx, uint16_t OcPolarity)	void TIM_ConfigOc6Polarity(TIM_Module* TIMx, uint16_t OcPolarity)	否	TIM 编号不一致。

通道 1 互补通道极性选择	void TIM_ConfigOc1NPolarity(TIM_Module* TIMx, uint16_t OcNPolarity)	void TIM_ConfigOc1NPolarity(TIM_Module* TIMx, uint32_t OcNPolarity)	否	TIM 编号不一致。
通道 2 互补通道极性选择	void TIM_ConfigOc2NPolarity(TIM_Module* TIMx, uint16_t OcNPolarity)	void TIM_ConfigOc2NPolarity(TIM_Module* TIMx, uint32_t OcNPolarity)	否	TIM 编号不一致。
通道 3 互补通道极性选择	void TIM_ConfigOc3NPolarity(TIM_Module* TIMx, uint16_t OcNPolarity)	void TIM_ConfigOc3NPolarity(TIM_Module* TIMx, uint32_t OcNPolarity)	否	TIM 编号不一致。
通道 4 互补通道极性选择	-	void TIM_ConfigOc4NPolarity(TIM_Module* TIMx, uint32_t OcNPolarity)	否	新增
捕获比较通道使能	void TIM_EnableCapCmpCh(TIM_Module* TIMx, uint16_t Channel, uint32_t TIM_CCx)	void TIM_EnableCapCmpCh(TIM_Module* TIMx, uint32_t Channel, uint32_t TIM_CCx)	否	TIM 编号不一致。 增加通道参数： TIM_CH_5 TIM_CH_6
捕获比较通道互补通道使能	void TIM_EnableCapCmpChN(TIM_Module* TIMx, uint16_t Channel, uint32_t TIM_CCxN)	void TIM_EnableCapCmpChN(TIM_Module* TIMx, uint32_t Channel, uint32_t TIM_CCxN)	否	TIM 编号不一致。 增加通道参数： TIM_CH_4
输出比较模式配置	void TIM_SelectOcMode(TIM_Module* TIMx, uint16_t Channel, uint16_t OcMode)	void TIM_SelectOcMode(TIM_Module* TIMx, uint32_t Channel, uint32_t OcMode)	否	TIM 编号不一致。 增加通道参数： TIM_CH_5 TIM_CH_6 增加 OCMODE 模式： TIM_OCMODE_OPMOD_RETRIG1 TIM_OCMODE_OPMOD_RETRIG2 TIM_OCMODE_COMBI_PWM1 TIM_OCMODE_COMBI_PWM2
更新事件使能	void TIM_EnableUpdateEvt(TIM_Module* TIMx, FunctionalState Cmd)	void TIM_EnableUpdateEvt(TIM_Module* TIMx, FunctionalState Cmd)	否	TIM 编号不一致。
更新请求源配置	void TIM_ConfigUpdateRequestIntSrc(TIM_Module* TIMx, uint16_t TIM_UpdateSource)	void TIM_ConfigUpdateRequestIntSrc(TIM_Module* TIMx, uint32_t TIM_UpdateSource)	否	TIM 编号不一致。
HALL 使能	void TIM_SelectHallSensor(TIM_Module* TIMx, FunctionalState Cmd)	void TIM_SelectHallSensor(TIM_Module* TIMx, FunctionalState Cmd)	否	TIM 编号不一致。
单脉冲模式配置	void TIM_SelectOnePulseMode(TIM_Module* TIMx, uint16_t TIM_OPMode)	void TIM_SelectOnePulseMode(TIM_Module* TIMx, uint32_t TIM_OPMode)	否	TIM 编号不一致。

TRGO 输出源选择	void TIM_SelectOutputTrig(TIM_Module* TIMx, uint16_t TIM_TRGOSource)	void TIM_SelectOutputTrig(TIM_Module* TIMx, uint32_t TIM_TRGOSource)	否	TIM 编号不一致。 增加 TRGO 信号源： TIM_TRGO_SRC_OC4_7_8_9REF
TRGO2 输出源选择	-	void TIM_SelectOutputTrig2(TIM_Module* TIMx, uint32_t TIM_TRGO2Source)	否	新增
从模式配置	void TIM_SelectSlaveMode(TIM_Module* TIMx, uint16_t TIM_SlaveMode)	void TIM_SelectSlaveMode(TIM_Module* TIMx, uint32_t TIM_SlaveMode)	否	TIM 编号不一致。 增加从模式参数： TIM_SLAVE_MODE_GATED_RESET TIM_SLAVE_MODE_TRIG_RESET
配置主从模式同步	void TIM_SelectMasterSlaveMode(TIM_Module* TIMx, uint16_t TIM_MasterSlaveMode)	void TIM_SelectMasterSlaveMode(TIM_Module* TIMx, uint32_t TIM_MasterSlaveMode)	否	TIM 编号不一致。
设置计数器	void TIM_SetCnt(TIM_Module* TIMx, uint16_t Counter)	void TIM_SetCnt(TIM_Module* TIMx, uint16_t Counter)	否	TIM 编号不一致。
设置 AR 值	void TIM_SetAutoReload(TIM_Module* TIMx, uint16_t Autoreload)	void TIM_SetAutoReload(TIM_Module* TIMx, uint32_t Autoreload)	否	TIM 编号不一致。
设置比较寄存器值 (CCDAT1)	void TIM_SetCmp1(TIM_Module* TIMx, uint16_t Compare1)	void TIM_SetCmp1(TIM_Module* TIMx, uint16_t Compare1)	否	TIM 编号不一致。
设置比较寄存器值 (CCDAT2)	void TIM_SetCmp2(TIM_Module* TIMx, uint16_t Compare2)	void TIM_SetCmp2(TIM_Module* TIMx, uint16_t Compare2)	否	TIM 编号不一致。
设置比较寄存器值 (CCDAT3)	void TIM_SetCmp3(TIM_Module* TIMx, uint16_t Compare3)	void TIM_SetCmp3(TIM_Module* TIMx, uint16_t Compare3)	否	TIM 编号不一致。
设置比较寄存器值 (CCDAT4)	void TIM_SetCmp4(TIM_Module* TIMx, uint16_t Compare4)	void TIM_SetCmp4(TIM_Module* TIMx, uint16_t Compare4)	否	TIM 编号不一致。
设置比较寄存器值 (CCDAT5)	void TIM_SetCmp5(TIM_Module* TIMx, uint16_t Compare5)	void TIM_SetCmp5(TIM_Module* TIMx, uint16_t Compare5)	否	TIM 编号不一致。
设置比较寄存器值 (CCDAT6)	void TIM_SetCmp6(TIM_Module* TIMx, uint16_t Compare6)	void TIM_SetCmp6(TIM_Module* TIMx, uint16_t Compare6)	否	TIM 编号不一致。
设置比较寄存器值 (CCDAT7)	-	void TIM_SetCmp7(TIM_Module* TIMx, uint16_t Compare7)	否	新增
设置比较寄存器值 (CCDAT8)	-	void TIM_SetCmp8(TIM_Module* TIMx, uint16_t Compare8)	否	新增
设置比较寄存器值 (CCDAT9)	-	void TIM_SetCmp9(TIM_Module* TIMx, uint16_t Compare9)	否	新增
设置比较寄存器值 (CCDDAT1)	-	void TIM_SetCmp1D(TIM_Module* TIMx, uint16_t compare1D)	否	新增

设置比较寄存器值(CCDDAT2)	-	void TIM_SetCmp2D(TIM_Module* TIMx, uint16_t compare2D)	否	新增
设置比较寄存器值(CCDDAT3)	-	void TIM_SetCmp3D(TIM_Module* TIMx, uint16_t compare3D)	否	新增
设置比较寄存器值(CCDDAT4)	-	void TIM_SetCmp4D(TIM_Module* TIMx, uint16_t compare4D)	否	新增
设置通道 1 输入捕获分频	void TIM_SetInCap1Prescaler(TIM_Module* TIMx, uint16_t TIM_ICPSC)	void TIM_SetInCap1Prescaler(TIM_Module* TIMx, uint16_t ICPrescaler)	否	TIM 编号不一致。
设置通道 2 输入捕获分频	void TIM_SetInCap2Prescaler(TIM_Module* TIMx, uint16_t TIM_ICPSC)	void TIM_SetInCap2Prescaler(TIM_Module* TIMx, uint16_t ICPrescaler)	否	TIM 编号不一致。
设置通道 3 输入捕获分频	void TIM_SetInCap3Prescaler(TIM_Module* TIMx, uint16_t TIM_ICPSC)	void TIM_SetInCap3Prescaler(TIM_Module* TIMx, uint16_t ICPrescaler)	否	TIM 编号不一致。
设置通道 4 输入捕获分频	void TIM_SetInCap4Prescaler(TIM_Module* TIMx, uint16_t TIM_ICPSC)	void TIM_SetInCap4Prescaler(TIM_Module* TIMx, uint16_t ICPrescaler)	否	TIM 编号不一致。
设置 ETR 输入源信号	-	void TIM_SelectETRInputSource(TIM_Module* TIMx, uint32_t TIM_ETRInputSource)	否	新增
设置 TIM 时钟分频	void TIM_SetClkDiv(TIM_Module* TIMx, uint16_t TIM_CKD)	void TIM_SetClkDiv(TIM_Module* TIMx, uint32_t TIM_CKD)	否	TIM 编号不一致。
获取比较寄存器值(CCDAT1)	uint16_t TIM_GetCap1(TIM_Module* TIMx)	uint16_t TIM_GetCap1(TIM_Module* TIMx)	否	TIM 编号不一致。
获取比较寄存器值(CCDAT2)	uint16_t TIM_GetCap2(TIM_Module* TIMx)	uint16_t TIM_GetCap2(TIM_Module* TIMx)	否	TIM 编号不一致。
获取比较寄存器值(CCDAT3)	uint16_t TIM_GetCap3(TIM_Module* TIMx)	uint16_t TIM_GetCap3(TIM_Module* TIMx)	否	TIM 编号不一致。
获取比较寄存器值(CCDAT4)	uint16_t TIM_GetCap4(TIM_Module* TIMx)	uint16_t TIM_GetCap4(TIM_Module* TIMx)	否	TIM 编号不一致。
获取比较寄存器值(CCDAT5)	uint16_t TIM_GetCap5(TIM_Module* TIMx)	uint16_t TIM_GetCap5(TIM_Module* TIMx)	否	TIM 编号不一致。
获取比较寄存器值(CCDAT6)	uint16_t TIM_GetCap6(TIM_Module* TIMx)	uint16_t TIM_GetCap6(TIM_Module* TIMx)	否	TIM 编号不一致。
获取比较寄存器值(CCDAT7)	-	uint16_t TIM_GetCap7(TIM_Module* TIMx)	否	新增
获取比较寄存器值(CCDAT8)	-	uint16_t TIM_GetCap8(TIM_Module* TIMx)	否	新增
获取比较寄存器值(CCDAT9)	-	uint16_t TIM_GetCap9(TIM_Module* TIMx)	否	新增

获取比较寄存器值(CCDDAT1)	-	uint16_t TIM_GetCap1D(TIM_Module* TIMx)	否	新增
获取比较寄存器值(CCDDAT2)	-	uint16_t TIM_GetCap2D(TIM_Module* TIMx)	否	新增
获取比较寄存器值(CCDDAT3)	-	uint16_t TIM_GetCap3D(TIM_Module* TIMx)	否	新增
获取比较寄存器值(CCDDAT4)	-	uint16_t TIM_GetCap4D(TIM_Module* TIMx)	否	新增
获取计数器值	uint32_t TIM_GetCnt(TIM_Module* TIMx)	uint32_t TIM_GetCnt(TIM_Module* TIMx)	否	TIM 编号不一致。
获取分频值	uint16_t TIM_GetPrescaler(TIM_Module* TIMx)	uint16_t TIM_GetPrescaler(TIM_Module* TIMx)	否	TIM 编号不一致。
获取 AR 值	uint16_t TIM_GetAutoReload(TIM_Module* TIMx)	uint16_t TIM_GetAutoReload(TIM_Module* TIMx)	否	TIM 编号不一致。
获取通道使能状态	FlagStatus TIM_GetCCENStatus(TIM_Module* TIMx, uint32_t TIM_CCEN)	FlagStatus TIM_GetCCENStatus(TIM_Module* TIMx, uint32_t TIM_CCEN)	否	TIM 编号不一致。 通道参数增加： TIM_CC4NEN
获取标志位状态	FlagStatus TIM_GetFlagStatus(TIM_Module* TIMx, uint32_t TIM_FLAG)	FlagStatus TIM_GetFlagStatus(TIM_Module* TIMx, uint32_t TIM_FLAG)	否	TIM 编号不一致。 标志位参数增加： TIM_FLAG_CC7 TIM_FLAG_CC8 TIM_FLAG_CC9 TIM_FLAG_BREAK2 TIM_FLAG_SYS_BREAK
清除标志位	void TIM_ClearFlag(TIM_Module* TIMx, uint32_t TIM_FLAG)	void TIM_ClearFlag(TIM_Module* TIMx, uint32_t TIM_FLAG)	否	TIM 编号不一致。 标志位参数增加： TIM_FLAG_CC7 TIM_FLAG_CC8 TIM_FLAG_CC9 TIM_FLAG_BREAK2 TIM_FLAG_SYS_BREAK
获取中断标志位	INTStatus TIM_GetIntStatus(TIM_Module* TIMx, uint32_t TIM_IT)	INTStatus TIM_GetIntStatus(TIM_Module* TIMx, uint32_t TIM_IT)	否	TIM 编号不一致。 标志位参数增加： TIM_INT_CC7 TIM_INT_CC8 TIM_INT_CC9 TIM_INT_BREAK2 TIM_INT_SYS_BREAK

清除中断标志位	void TIM_ClrIntPendingBit(TIM_Module* TIMx, uint32_t TIM_IT)	void TIM_ClrIntPendingBit(TIM_Module* TIMx, uint32_t TIM_IT)	否	TIM 编号不一致。 标志位参数增加： TIM_INT_CC7 TIM_INT_CC8 TIM_INT_CC9 TIM_INT_BREAK2 TIM_INT_SYS_BREAK
设置捕获通道 1 的极性，输入选择，滤波	static void ConfigTI1(TIM_Module* TIMx, uint16_t IcPolarity, uint16_t IcSelection, uint16_t IcFilter)	void ConfigTI1(TIM_Module* TIMx, uint32_t ICPolarity, uint32_t ICSelection, uint32_t ICFilter)	否	TIM 编号不一致。
设置捕获通道 2 的极性，输入选择，滤波	static void ConfigTI2(TIM_Module* TIMx, uint16_t IcPolarity, uint16_t IcSelection, uint16_t IcFilter)	void ConfigTI2(TIM_Module* TIMx, uint32_t ICPolarity, uint32_t ICSelection, uint32_t ICFilter)	否	TIM 编号不一致。
设置捕获通道 3 的极性，输入选择，滤波	static void ConfigTI3(TIM_Module* TIMx, uint16_t IcPolarity, uint16_t IcSelection, uint16_t IcFilter)	void ConfigTI3(TIM_Module* TIMx, uint32_t ICPolarity, uint32_t ICSelection, uint32_t ICFilter)	否	TIM 编号不一致。
设置捕获通道 4 的极性，输入选择，滤波	static void ConfigTI4(TIM_Module* TIMx, uint16_t IcPolarity, uint16_t IcSelection, uint16_t IcFilter)	void ConfigTI4(TIM_Module* TIMx, uint32_t ICPolarity, uint32_t ICSelection, uint32_t ICFilter)	否	TIM 编号不一致。
使能中心对齐非对称模式	-	void TIM_AsymmetricEnable(TIM_Module* TIMx, FunctionalState Cmd)	否	新增
使能 4/7/8/9 通道触发 ADC	-	void TIM_OCxRefTriggerADC(TIM_Module* TIMx, uint32_t OCxRef, FunctionalState Cmd)	否	新增
通道 1 输入数字滤波配置	-	void TIM_IC1FiltConfig(TIM_Module* TIMx, TIM_FiltInitType* TIM_FiltInitStruct)	否	新增
通道 2 输入数字滤波配置	-	void TIM_IC2FiltConfig(TIM_Module* TIMx, TIM_FiltInitType* TIM_FiltInitStruct)	否	新增
通道 3 输入数字滤波配置	-	void TIM_IC3FiltConfig(TIM_Module* TIMx, TIM_FiltInitType* TIM_FiltInitStruct)	否	新增
通道 4 输入数字滤波配置	-	void TIM_IC4FiltConfig(TIM_Module* TIMx, TIM_FiltInitType* TIM_FiltInitStruct)	否	新增
通道 1 输入数字滤波使能	-	void TIM_IC1FiltEnable(TIM_Module* TIMx,FunctionalState Cmd)	否	新增
通道 2 输入数字滤波使能	-	void TIM_IC2FiltEnable(TIM_Module* TIMx,FunctionalState Cmd)	否	新增
通道 3 输入数字滤波使能	-	void TIM_IC3FiltEnable(TIM_Module* TIMx,FunctionalState Cmd)	否	新增

通道 4 输入数字滤波使能	-	void TIM_IC4FiltEnable(TIM_Module* TIMx,FunctionalState Cmd)	否	新增
通道 x 输入数字滤波器输出状态	-	FlagStatus TIM_GetFiltStatus(TIM_Module* TIMx, uint32_t TIM_FiltFlag)	否	新增

4.3.28 ADC

下表是N32G45x系列和N32H47x_48x系列在ADC模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用ADC模块驱动API函数可以依照下表做替换；

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
ADC 复位	void ADC_DeInit(ADC_Module* ADCx)	void ADC_DeInit(ADC_Module* ADCx)	是	无
ADC 初始化	void ADC_Init(ADC_Module* ADCx, ADC_InitType* ADC_InitStruct)	ADC_Init(ADC_Module* ADCx, ADC_InitType* ADC_InitStruct)	是	新增 Resolution 配置参数
ADC 结构体初始化	void ADC_InitStruct(ADC_InitType* ADC_InitStruct)	void ADC_InitStruct(ADC_InitType* ADC_InitStruct)	是	新增 Resolution 配置参数
ADC 模块使能	void ADC_Enable(ADC_Module* ADCx, FunctionalState Cmd)	void ADC_Enable(ADC_Module* ADCx, FunctionalState Cmd)	是	无
ADC DMA 使能	void ADC_EnableDMA(ADC_Module* ADCx, FunctionalState Cmd)	void ADC_SetDMATransferMode(ADC_Module *ADCx, uint32_t DMAMode)	否	新增了 ADC DMA 模式设置的功能，输入参数 ADC_MULTI_REG_DMA_EACH_ADC 等同于 ADC_EnableDMA(ADCx,ENABLE)
ADC 中断配置	void ADC_ConfigInt(ADC_Module* ADCx, uint16_t ADC_IT, FunctionalState Cmd)	void ADC_ConfigInt(ADC_Module* ADCx, uint16_t ADC_IT, FunctionalState Cmd)	是	新增参数： ADC_INT_ENDCA ADC_INT_JENDCA ADC_INT_ENDCERR ADC_INT_RDY ADC_INT_PDRDY ADC_INT_EOSAMP
ADC 校准	void ADC_StartCalibration(ADC_Module* ADCx)	void ADC_CalibrationOperation(ADC_Module* ADCx, ADC_CALI_MOD cali_mode)	否	新增参数：cali_mode 新版区分单端模式校准和差分模式校准 ADC_StartCalibration(ADCx)等同于 ADC_CalibrationOperation(ADCx

				, ADC_CALIBRATION_SIGNAL_MODE)
获取 ADC 校准状态	FlagStatus ADC_GetCalibrationStatus(ADC_Module* ADCx)	FlagStatus ADC_GetCalibrationStatus(ADC_Module* ADCx)	是	无
复位 ADC 校准	无	void ADC_CalibrationReset(ADC_Module* ADCx, ADC_CALI_MOD cali_mode)	否	新增功能
开启规则通道软件转换	void ADC_EnableSoftwareStartConv(ADC_Module* ADCx, FunctionalState Cmd)	void ADC_EnableSoftwareStartConv(ADC_Module* ADCx, FunctionalState Cmd)	是	无
获取规则通道软件开始转换状态	FlagStatus ADC_GetSoftwareStartConvStatus(ADC_Module* ADCx)	FlagStatus ADC_GetSoftwareStartConvStatus(ADC_Module* ADCx);	是	无
开启注入通道软件转换	void ADC_EnableSoftwareStartInjectedConv(ADC_Module* ADCx, FunctionalState Cmd);	void ADC_EnableSoftwareStartInjectedConv(ADC_Module* ADCx, FunctionalState Cmd);	是	无
获取注入通道软件开始转换状态	FlagStatus ADC_GetSoftwareStartInjectedConvCmdStatus(ADC_Module* ADCx);	FlagStatus ADC_GetSoftwareStartInjectedConvCmdStatus(ADC_Module* ADCx);	是	无
配置不连续模式子组数	void ADC_ConfigDiscModeChannelCount(ADC_Module* ADCx, uint8_t Number)	void ADC_ConfigDiscModeChannelCount(ADC_Module* ADCx, uint8_t Number);	是	无
使能规则通道不连续模式	void ADC_EnableDiscMode(ADC_Module* ADCx, FunctionalState Cmd)	void ADC_EnableDiscMode(ADC_Module* ADCx, FunctionalState Cmd);	是	无
使能注入通道不连续模式	void ADC_EnableInjectedDiscMode(ADC_Module* ADCx, FunctionalState Cmd);	void ADC_EnableInjectedDiscMode(ADC_Module* ADCx, FunctionalState Cmd);	是	无
规则组通道配置	void ADC_ConfigRegularChannel(ADC_Module* ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime)	void ADC_ConfigRegularChannel(ADC_Module* ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime)	是	无
自动注入使能	void ADC_EnableAutoInjectedConv(ADC_Module* ADCx, FunctionalState Cmd)	void ADC_EnableAutoInjectedConv(ADC_Module* ADCx, FunctionalState Cmd)	是	无
规则通道外部触发使能	void ADC_EnableExternalTrigConv(ADC_Module* ADCx, FunctionalState Cmd)	void ADC_SetRegularTriggerEdge(ADC_Module* ADCx, uint32_t ExternalRegularTriggerEdge)	否	新增外部触发有效边沿配置 ADC_EnableAutoInjectedConv(ADCx,ENABLE)与 ADC_SetRegularTriggerEdge(ADCx，ADC_REG_TRIG_EXT_RISING)相同
注入通道外部触发使能	void ADC_EnableExternalTrigInjectedConv(ADC_Module* ADCx, FunctionalState Cmd)	void ADC_SetInjectTriggerEdge(ADC_Module* ADCx, uint32_t ExternalInjectTriggerEdge)	否	新增外部触发有效边沿配置 ADC_ EnableExternalTrigInjectedConv (ADCx,ENABLE)与 ADC_

				SetInjectTriggerEdge (ADCx，ADC_INJ_TRIG_EXT_RISING)相同
获取单 ADC 转换结果	uint16_t ADC_GetDat(ADC_Module* ADCx)	uint16_t ADC_GetDat(ADC_Module* ADCx)	是	无
获取多 ADC 转换结果	uint32_t ADC_GetDualModeConversionDat(ADC_Module* ADCx)	uint32_t ADC_GetMutiModeConversionDat(ADC_Module* ADCx)	否	仅修改函数名
配置规则通道的外部触发源	无	void ADC_ConfigExternalTrigRegularConv(ADC_Module* ADCx, uint32_t ADC_ExternalTrigRegularConv)	否	新增函数
配置注入通道的外部触发源	void ADC_ConfigExternalTrigInjectedConv(ADC_Module* ADCx, uint32_t ADC_ExternalTrigInjecConv)	void ADC_ConfigExternalTrigInjectedConv(ADC_Module* ADCx, uint32_t ADC_ExternalTrigInjecConv)	是	无
注入组通道配置	void ADC_ConfigInjectedChannel(ADC_Module* ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime)	void ADC_ConfigInjectedChannel(ADC_Module* ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime)	是	无
配置注入序列长度	void ADC_ConfigInjectedSequencerLength(ADC_Module* ADCx, uint8_t Length)	void ADC_ConfigInjectedSequencerLength(ADC_Module* ADCx, uint8_t Length)	是	无
偏移数据设置	void ADC_SetInjectedOffsetDat(ADC_Module* ADCx, uint8_t ADC_InjectedChannel, uint16_t Offset)	void ADC_SetOffsetConfig(ADC_Module* ADCx, uint8_t ADC_Offset, ADC_OffsetType* ADC_OffsetStruct)	否	功能修改参考用户手册描述 支持注入通道和规则通道的偏移，入参改动比较大，建议先了解功能
获取偏移配置	无	void ADC_GetOffsetConfig(ADC_Module* ADCx, uint8_t ADC_Offset, ADC_OffsetType* ADC_OffsetStruct)	否	新增函数
获取注入通道转换数据	uint16_t ADC_GetInjectedConversionDat(ADC_Module* ADCx, uint8_t ADC_InjectedChannel)	uint16_t ADC_GetInjectedConversionDat(ADC_Module* ADCx, uint8_t ADC_InjectedChannelOffset)	是	入参方式有修改： ADC_INJ_CH_1 对应 ADC_INJECT_DATA_OFFSET_1 …… ADC_INJ_CH_4 对应 ADC_INJECT_DATA_OFFSET_4
配置模拟看门狗的工作模式	void ADC_ConfigAnalogWatchdogWorkChannelType(ADC_Module* ADCx, uint32_t ADC_AnalogWatchdog)	void ADC_ConfigAnalogWatchdog1WorkChannelType(ADC_Module* ADCx, uint32_t ADC_AnalogWatchdog)	是	无
设置看门狗监视的通道	void ADC_ConfigAnalogWatchdogSingleChannel(ADC_Module* ADCx, uint8_t ADC_Channel)	void ADC_ConfigAnalogWatchdog1SingleChannel(ADC_Module* ADCx, uint8_t ADC_Channel)	是	无

设置看门狗阈值	void ADC_ConfigAnalogWatchdogThresholds(ADC_Module* ADCx, uint16_t HighThreshold, uint16_t LowThreshold)	void ADC_ConfigAnalogWatchdogThresholds(ADC_Module* ADCx, ADC_AWDG Awdg, uint16_t HighThreshold, uint16_t LowThreshold)	否	入参有变化； 原函数仅配置看门狗 1，新的函数可以选择配置看门狗 1/2/3
使能温度传感器与内置参考电压	void ADC_EnableTempSensorVrefint(FunctionalState Cmd)	void ADC_EnableTempSensorVrefint(FunctionalState Cmd)	是	无
获取 ADC 状态	FlagStatus ADC_GetFlagStatus(ADC_Module* ADCx, uint8_t ADC_FLAG)	FlagStatus ADC_GetFlagStatus(ADC_Module* ADCx, uint16_t ADC_FLAG)	是	入参新增以下： - ADC_FLAG_EOC_ANY - ADC_FLAG_JEOC_ANY - ADC_FLAG_ENDC_ERROR - ADC_FLAG_RDY - ADC_FLAG_PDRDY - ADC_FLAG_EOSAMP - ADC_FLAG_TCFLAG
清除 ADC 状态	void ADC_ClearFlag(ADC_Module* ADCx, uint8_t ADC_FLAG)	void ADC_ClearFlag(ADC_Module* ADCx, uint16_t ADC_FLAG)	是	无
获取中断状态	INTStatus ADC_GetIntStatus(ADC_Module* ADCx, uint16_t ADC_IT)	INTStatus ADC_GetIntStatus(ADC_Module* ADCx, uint16_t ADC_IT)	是	入参新增以下： - ADC_ INT_EOC_ANY - ADC_ INT_JEOC_ANY - ADC_ INT_ENDC_ERROR - ADC_ INT_RDY - ADC_ INT_PDRDY - ADC_ INT_EOSAMP - ADC_ INT_TCFLAG
清除中断状态	void ADC_ClearIntPendingBit(ADC_Module* ADCx, uint16_t ADC_IT)	void ADC_ClearIntPendingBit(ADC_Module* ADCx, uint16_t ADC_IT)	是	入参新增以下： - ADC_ INT_EOC_ANY - ADC_ INT_JEOC_ANY - ADC_ INT_ENDC_ERROR - ADC_ INT_RDY - ADC_ INT_PDRDY - ADC_ INT_EOSAMP - ADC_ INT_TCFLAG
设置通道的差分模式或者单端模式	void ADC_SetDifChs(ADC_Module* ADCx,uint32_t DifChs)	void ADC_SetChannelSingleDiff(ADC_Module* ADCx, uint32_t Channel, uint32_t SingleDiff)	否	修改函数的形参，具体参考函数的参数枚举
ADC 附加功能设置	void ADC_InitEx(ADC_Module* ADCx, ADC_InitTypeEx* ADC_InitStructEx)	void ADC_SetConvResultBitNum(ADC_Module* ADCx, uint32_t ResultBitNum)	否	删除该函数，并拆分这个函数的功能
获取 ADC ready 状态	FlagStatus ADC_GetFlagStatusNew(ADC_Module* ADCx, uint8_t ADC_FLAG_NEW)	无	否	该函数与 ADC_GetFlagStatus 合并

设置 ADC bypass 模式是否使能	void ADC_SetBypassCalibration(ADC_Module* ADCx, FunctionalState en)	void ADC_SetBypassCalibration(ADC_Module* ADCx, FunctionalState Cmd)	是	无
设置分辨率	void ADC_SetConvResultBitNum(ADC_Module* ADCx, uint32_t ResultBitNum)	void ADC_SetConvResultBitNum(ADC_Module* ADCx, uint32_t ResultBitNum)	是	无
设置 ADC 的时钟模式	void ADC_AHB_Clock_Mode_Config(ADC_Module* ADCx) void ADC_PLL_Clock_Mode_Config(ADC_Module* ADCx)	void ADC_SelectClockMode(ADC_Module* ADCx, uint32_t ClockMode)	否	优化函数的写法
设置 ADC 时钟模式以及分频	void ADC_ConfigClk(ADC_CTRL3_CKMOD ADC_ClkMode, uint32_t RCC_ADCHCLKPrescaler)	void ADC_ConfigClk(ADC_CTRL3_CKMOD ADC_ClkMode, uint32_t RCC_ADCHCLKPrescaler)	是	无
设置多 ADC 交叉模式的延时时间	无	void ADC_SetMultiTwoSamplingDelay(ADC_Module* ADCx, uint32_t MultiTwoSamplingDelay)	否	新增函数
停止注入通道转换	无	void ADC_StopInjectedConv(ADC_Module* ADCx)	否	新增函数
停止规则通道转换	无	void ADC_StopRegularConv(ADC_Module* ADCx)	否	新增函数
获取增益补偿使能状态	无	FlagStatus ADC_GetGainCompensationCmdStatus(ADC_Module* ADCx)	否	新增函数
设置看门狗 1 的滤波个数	无	void ADC_SetAWDG1FilteringConfig(ADC_Module *ADCx, uint32_t FilteringCount)	否	新增函数
设置看门狗 2/3 的监视通道组	无	void ADC_SetAnalogWatchdog23MonitChannels(ADC_Module* ADCx, uint8_t AWDG_RegEnOffset, uint32_t AWDG_ChannelGroup)	否	新增函数
获取看门狗 2/3 的监视通道组	无	uint32_t ADC_GetAnalogWatchdog23MonitChannels(ADC_Module* ADCx, uint8_t AWDG_RegEnOffset)	否	新增函数
设置看门狗 2/3 的中断使能通道	无	void ADC_SetAnalogWatchdog23IntConfig(ADC_Module* ADCx, uint8_t AWDG_RegIntEnOffset, uint32_t AWDG_ChannelEn)	否	新增函数
获取看门狗 2/3 的中断使能通道	无	uint32_t ADC_GetAnalogWatchdog23IntConfig(ADC_Module* ADCx, uint8_t AWDG_RegEnOffset)	否	新增函数
获取看门狗 2/3 的状态标志	无	uint32_t ADC_GetAnalogWatchdog23StatusFlag(ADC_Module* ADCx, uint8_t AWDG_RegSTSOffset)	否	新增函数
清除看门狗 2/3 的状态标志	无	void ADC_ClearAnalogWatchdog23StatusFlag(ADC_Module	否	新增函数

		* ADCx, uint8_t AWDG_RegSTSOffset, uint32_t AWDG_ChannelFlag)		
设置 ADC 校准系数	无	void ADC_SetCalibrationFactor(ADC_Module *ADCx, uint32_t SingleDiff, uint32_t CalibrationFactor)	否	新增函数
设置增益补偿使能以及增益补偿值	无	void ADC_SetGainCompensation(ADC_Module *ADCx, uint32_t GainCompensationValue)	否	新增函数
使能不连续的过采样模式	无	void ADC_EnableOverSamplingDiscont(ADC_Module *ADCx, FunctionalState Cmd)	否	新增函数
设置过采样的工作范围	无	void ADC_SetOverSamplingScope(ADC_Module *ADCx, uint32_t OversampleScope)	否	新增函数
配置过采样倍率和右移位数	无	void ADC_ConfigOverSamplingRatioAndShift(ADC_Module *ADCx, uint32_t Ratio, uint32_t Shift)	否	新增函数
使能深度睡眠模式	无	void ADC_EnableDeepSleepMode(ADC_Module* ADCx, FunctionalState Cmd)	否	新增函数
使能电池电压	无	void ADC_EnableBatteryVoltageMonitor(ADC_Module* ADCx, FunctionalState Cmd)	否	新增函数
是能校准值自动载入	无	void ADC_EnableCalibrationAutoLoad(ADC_Module* ADCx, FunctionalState Cmd)	否	新增函数
使能通道 1 正端是否连接到 PGA	无	void ADC_EnableCH1PositiveEndConnetPGA_P(ADC_Module *ADCx, FunctionalState Cmd)	否	新增函数
使能通道 1 负端是否连接到 PGA	无	void ADC_EnableCH1NegtiveEndConnetPGA_N(ADC_Module *ADCx, FunctionalState Cmd)	否	新增函数
使能通道 2 正端是否连接到 PGA	无	void ADC_EnableCH2PositiveEndConnetPGA_P(ADC_Module *ADCx, FunctionalState Cmd)	否	新增函数
使能 ADC FIFO 功能	无	void ADC_EnableFIFO(ADC_Module* ADCx, FunctionalState Cmd)	否	新增函数
清除 FIFO 数据	无	void ADC_ClearFIFO(ADC_Module* ADCx)	否	新增函数
获取 FIFO 水线	无	void ADC_ConfigFIFOWaterLevel(ADC_Module* ADCx, uint32_t FIFO_Level)	否	新增函数
获取 FIFO 有效个数	无	uint8_t ADC_GetFIFOInvalidedDataCount(ADC_Module* ADCx)	否	新增函数
获取 FIFO 状态标志	无	FlagStatus ADC_GetFIFOFlagStatus(ADC_Module* ADCx, uint16_t ADC_FIFOFLAG)	否	新增函数

清除 FIFO 状态标志	无	void ADC_ClearFIFOFlag(ADC_Module* ADCx, uint16_t ADC_FIFO_FLAG)	否	新增函数
配置 ADC FIFO 中断	无	void ADC_ConfigFIFOInt(ADC_Module* ADCx, uint16_t ADC_FIFO_IT, FunctionalState Cmd)	否	新增函数
清除 FIFO 中断状态	无	void ADC_ClearFIFOIntPendingBit(ADC_Module* ADCx, uint16_t ADC_FIFO_IT)	否	新增函数
选择具体 EXTI 线作为 ADC 规则通道的触发源	无	void ADC_ConfigRegularExtLineTrigSource(ADC_Module* ADCx, uint32_t ADC_trigger)	否	新增函数
选择具体 EXTI 线作为 ADC 注入通道的触发源	无	void ADC_ConfigInjectedExtLineTrigSource(ADC_Module* ADCx, uint32_t ADC_trigger)	否	新增函数
使能 ADC 的校准值自动加载功能	无	void ADC_EnableCalibrationAutoLoad(ADC_Module* ADCx, FunctionalState Cmd)	否	新增函数
设置 ADC 工作电源	无	void ADC_ConfigADCPowerSupport(ADC_Module* ADCx, uint32_t ADC_PowerSupportSel)	否	新增函数

下图是N32H47x_48x系列在ADC模块的新增功能比较多，包括过采样，偏移补偿，增益补偿，看门狗2/3的全通道监视，exti线触发等：

1. EXTI 线触发的 Demo 差异比较大，EXTI 线的选择参考下图所示：

```c
*\*\return  none
**/
void ADC_Initial(void)
{
    /* ADC1 configuration ------------------------------------------------------*/
    ADC_InitStructure.WorkMode       = ADC_WORKMODE_INDEPENDENT;
    ADC_InitStructure.MultiChEn      = ENABLE;
    ADC_InitStructure.ContinueConvEn = DISABLE;
    ADC_InitStructure.ExtTrigSelect  = ADC_EXT_TRIG_REG_CONV_EXT_INT0_15;
    ADC_InitStructure.DatAlign       = ADC_DAT_ALIGN_R;
    ADC_InitStructure.ChsNumber      = 2;
    ADC_InitStructure.Resolution     = ADC_DATA_RES_12BIT;
    ADC_Init(ADC1, &ADC_InitStructure);
    /* ADC1 regular channel4 12 configuration */
    ADC_ConfigRegularChannel(ADC1, ADC1_Channel_04_PA1, 1, ADC_SAMP_TIME_CYCLES_239_5);
    ADC_ConfigRegularChannel(ADC1, ADC1_Channel_12_PB1, 2, ADC_SAMP_TIME_CYCLES_239_5);

    /* Regular discontinuous mode channel number configuration */
    ADC_ConfigDiscModeChannelCount(ADC1, 1);
    /* Enable regular discontinuous mode */
    ADC_EnableDiscMode(ADC1, ENABLE);

    /* Set injected sequencer length */
    ADC_ConfigInjectedSequencerLength(ADC1, 2);
    /* ADC1 injected channel configuration */
    ADC_ConfigInjectedChannel(ADC1, ADC1_Channel_14_PB11, 1, ADC_SAMP_TIME_CYCLES_239_5);
    ADC_ConfigInjectedChannel(ADC1, ADC1_Channel_11_PB12, 2, ADC_SAMP_TIME_CYCLES_239_5);
    /* ADC1 injected external trigger configuration */
    ADC_ConfigExternalTrigInjectedConv(ADC1, ADC_EXT_TRIG_INJ_CONV_EXT_INT0_15);

    /* Enable JENDC interrupt */
    ADC_ConfigInt(ADC1, ADC_INT_JENDC, ENABLE);

    /* Enable ADC1 DMA */
    ADC_SetDMATransferMode(ADC1, ADC_MULTI_REG_DMA_EACH_ADC);

    ADC_ConfigRegularExtLineTrigSource(ADC1,Regular_ExtLine_TrigSource);
    ADC_ConfigInjectedExtLineTrigSource(ADC1,Injected_ExtLine_TrigSource);

    /* Enable ADC1 */
    ADC_Enable(ADC1, ENABLE);
    /*Check ADC Ready*/
    while(ADC_GetFlagStatus(ADC1,ADC_FLAG_RDY) == RESET)
        ;
    /* Start ADC1 calibration */
    ADC_CalibrationOperation(ADC1,ADC_CALIBRATION_SIGNAL_MODE);
    /* Check the end of ADC1 calibration */
    while (ADC_GetCalibrationStatus(ADC1))
        ;
}
```

```c
/**
 * @brief  Configures the different EXTI lines.
 */
void EXTI_Configuration(void)
{
    EXTI_InitType EXTI_InitStructure;

    GPIO_ConfigEXTILine(GPIOE_PORT_SOURCE, GPIO_PIN_SOURCE11);
    /* EXTI line11 configuration -----------------------------------------------*/
    EXTI_InitStructure.EXTI_Mode    = EXTI_Mode_Event;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
    EXTI_InitStructure.EXTI_Line    = EXTI_LINE11;
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    EXTI_InitPeripheral(&EXTI_InitStructure);

    /* Select the EXTI Line15 the GPIO pin source */
    GPIO_ConfigEXTILine(GPIOE_PORT_SOURCE, GPIO_PIN_SOURCE15);
    /* EXTI line15 configuration -----------------------------------------------*/
    EXTI_InitStructure.EXTI_Mode    = EXTI_Mode_Event;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
    EXTI_InitStructure.EXTI_Line    = EXTI_LINE15;
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    EXTI_InitPeripheral(&EXTI_InitStructure);
}

/**
```

2. 其他的新增功能，请参考具体的 Demo：

4.3.29**DAC**

下表是N32G45x系列和N32H47x_48x系列在DAC模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用DAC模块驱动API函数可以依照下表做替换；

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
DAC 复位	void DAC_DeInit(void)	void DAC_DeInit(DACX DACx)	否	新增 DACx 的形参传入
DAC 初始化	void DAC_Init(uint32_t DAC_Channel, DAC_InitType* DAC_InitStruct)	void DAC_Init(DACX DACx, DAC_InitType* DAC_InitStruct)	否	由于 DAC 数量增加导致传入的形参由 DAC_Channel 变更到 DACx， 形参 DAC_InitStruct 新增了成员变量 DAC_ConnectExternalPin DAC_ConnectOnChipPeripheral DAC_SignedFormat DAC_DMADoubleDataMode DAC_TriggerEnable 等一些差异，具体参考驱动代码
DAC 结构体初始化	void DAC_ClearStruct(DAC_InitType* DAC_InitStruct)	void DAC_StructInit(DAC_InitType* DAC_InitStruct)	否	形参 DAC_InitStruct 新增了成员变量 DAC_ConnectExternalPin DAC_ConnectOnChipPeripheral DAC_SignedFormat DAC_DMADoubleDataMode DAC_TriggerEnable 等一些差异，具体参考驱动代码
DAC 使能	void DAC_Enable(uint32_t DAC_Channel, FunctionalState Cmd)	void DAC_Enable(DACX DACx, FunctionalState Cmd)	否	由于 DAC 数量增加导致传入的形参由 DAC_Channel 变更到 DACx，
DAC DMA 使能	void DAC_DmaEnable(uint32_t DAC_Channel, FunctionalState Cmd)	void DAC_DmaEnable(DACX DACx, FunctionalState Cmd)	否	由于 DAC 数量增加导致传入的形参由 DAC_Channel 变更到 DACx，
DAC 软件触发使能	void DAC_SoftTrgEnable(uint32_t DAC_Channel, FunctionalState Cmd)	void DAC_SoftTrgEnable(DACX DACx, FunctionalState Cmd)	否	由于 DAC 数量增加导致传入的形参由 DAC_Channel 变更到 DACx，
DAC 两路同时软件触发使能	void DAC_DualSoftwareTrgEnable(FunctionalState Cmd)	void DAC_DualSoftwareTrgEnable(DAC_Module *Dual_DACx, FunctionalState Cmd)	否	由于 DAC 数量导致新增形参 Dual_DACx，
DAC 锯齿波步进触发使能	无	void DAC_SoftTrgSawStepEnable(DACX DACx, FunctionalState Cmd)	否	函数新增
DAC 锯齿波两路同时步进触发使能	无	void DAC_DualSoftwareTrgSawStepEnable(DAC_Module *Dual_DACx, FunctionalState Cmd)	否	函数新增
DAC 波形生成配置	void DAC_WaveGenerationEnable(uint32_t DAC_Channel, uint32_t DAC_Wave, FunctionalState Cmd)	void DAC_WaveGenerationConfig(DACX DACx, uint32_t DAC_Wave)	否	由于 DAC 数量增加导致传入的形参由 DAC_Channel 变更到 DACx， 以及函数优化
DAC 保持寄存器写入	void DAC_SetCh1Data(uint32_t DAC_Align, uint16_t Data) void DAC_SetCh2Data(uint32_t DAC_Align, uint16_t Data)	void DAC_SetData(DACX DACx, uint32_t DAC_Align, uint16_t Data)	否	函数优化

DAC 双通道保持寄存器同时写入	void DAC_SetDualChData(uint32_t DAC_Align, uint16_t Data2, uint16_t Data1)	void DAC_SetDualChData(DAC_Module *Dual_DACx, uint32_t DAC_Align, uint16_t Data2, uint16_t Data1)	否	由于 DAC 数量导致新增形参 Dual_DACx，
获取 DAC 输出数据寄存器的值	uint16_t DAC_GetOutputDataVal(uint32_t DAC_Channel)	uint16_t DAC_GetOutputDataVal(DACX DACx)	否	由于 DAC 数量增加导致传入的形参由 DAC_Channel 变更到 DACx， 以及函数优化
设置锯齿波复位值	无	void DAC_SetSawtoothResetValue(DACX DACx, uint16_t ResetValue)	否	函数新增
设置锯齿波步进值	无	void DAC_SetSawtoothStepValue(DACX DACx, uint16_t StepData)	否	函数新增
使能 DAC 校准	无	void DAC_CaliEnable(DACX DACx, FunctionalState Cmd)	否	函数新增
使能 DAC 输出连接到芯片内部	无	void DAC_ConnetToOnChipEnable(DACX DACx, FunctionalState Cmd)	否	函数新增
使能 DAC 输出连接到外部 IO	无	void DAC_ConnetToExternalPinEnable(DACX DACx, FunctionalState Cmd)	否	函数新增
使能 DAC DMA 双数据模式	无	void DAC_DMADoubleDataModeEnable(DACX DACx, FunctionalState Cmd)	否	函数新增
使能有符号格式输出	无	void DAC_SignFormatModeEnable(DACX DACx, FunctionalState Cmd)	否	函数新增
使能输出高驱动能力	无	void DAC_HighDriveAbilityEnable(DACX DACx, FunctionalState Cmd)	否	函数新增
获取 DAC 状态	无	FlagStatus DAC_GetFlagSts(DACX DACx, uint32_t DAC_FLAG)	否	函数新增
清除 DAC 状态位	无	void DAC_ClearFlag(DACX DACx, uint32_t DAC_FLAG)	否	函数新增
DAC 中断配置	无	void DAC_ConfigInt(DACX DACx, uint32_t DAC_IT, FunctionalState Cmd)	否	函数新增
获取 DAC 中断状态标志位	无	FlagStatus DAC_GetIntSts(DACX DACx, uint32_t DAC_IT)	否	函数新增
清除 DAC 中断状态标志位	无	void DAC_ClearITPendingBit(DACX DACx, uint32_t DAC_IT)	否	函数新增
DAC 分频系数配置	无	void DAC_ConfigClkPrescaler(DAC_Module* DACx, uint8_t Prescaler)	否	函数新增
设置 DAC 高频率工作模式	无	void DAC_SetHighFrequencyMode(DAC_Module* DACx, uint32_t mode)	否	函数新增
DAC 设置用户校准值	无	void DAC_SetUserTrimming(DACX DACx, uint8_t TrimmingValue)	否	函数新增

通过 DACx 获取基地址	无	static DAC_Module* DAC_BaseAddrGet(DACX DACx)	否	函数新增 注：此函数仅供.C 调用

下图是N32H47x_48x系列在DAC模块的新增功能比较多，包括DAC对内/对外输出配置，DAC锯齿波生成，DMA双数据模式，DAC中断，DAC校准等功能：

1. 下图的左边是 G45x 的输出两路三角波到 I/O 的配置方式，右边是 N32H47x_48x 系列的配置方式。除了以下的区别外，N32H47x_48x 系列还需要新增 DAC 分频系数配置（DAC_ConfigClkPrescaler() ）以及 DAC 的频率模式（DAC_SetHighFrequencyMode() ）配置，具体参考“TwoDACsTriangleWave”的 Demo 例程；

```
/**
  * @brief  DAC channel1 Configuration.
  */
void DAC_ChannelsConfig(void)
{
    DAC_InitType DAC_InitStructure;

    /* DAC Periph clock enable */
    RCC_EnableAPB1PeriphClk(RCC_APB1_PERIPH_DAC, ENABLE);
    /* DAC channel1 Configuration */
    DAC_InitStructure.Trigger          = DAC_TRG_T5_TRGO;
    DAC_InitStructure.WaveGen          = DAC_WAVEGEN_TRIANGLE;
    DAC_InitStructure.LfsrUnMaskTriAmp = DAC_TRIAMP_2047;
    DAC_InitStructure.BufferOutput     = DAC_BUFFOUTPUT_DISABLE;
    DAC_Init(DAC_CHANNEL_1, &DAC_InitStructure);

    /* DAC channel2 Configuration */
    DAC_InitStructure.LfsrUnMaskTriAmp = DAC_TRIAMP_1023;
    DAC_Init(DAC_CHANNEL_2, &DAC_InitStructure);

    /* Enable DAC Channel1: Once the DAC channel1 is enabled, PA.04 is
            automatically connected to the DAC converter. */
    DAC_Enable(DAC_CHANNEL_1, ENABLE);

    /* Enable DAC Channel2: Once the DAC channel2 is enabled, PA.05 is
            automatically connected to the DAC converter. */
    DAC_Enable(DAC_CHANNEL_2, ENABLE);

    /* Set DAC dual channel DR12DCH register */
    DAC_SetDualChData(DAC_ALIGN_R_12BIT, 0x100, 0x100);
}
```

```
/**
 *\*\name    DAC_Configuratoin.
 *\*\fun     Configures the DAC1 module.
 *\*\return  none
 **/
void DAC_Configuratoin(void)
{
    DAC_InitType DAC_InitStructure;

    DAC_StructInit(&DAC_InitStructure);
    /* DAC1 Configuration */
    DAC_InitStructure.DAC_Trigger                      = DAC_Trigger_ATIM1_TRGO;
    DAC_InitStructure.DAC_WaveGeneration               = DAC_WaveGeneration_Triangle;
    DAC_InitStructure.DAC_LFSRUnmask_TriangleAmplitude = DAC_TriangleAmplitude_2047;
    DAC_InitStructure.DAC_OutputBuffer                 = DISABLE;
    DAC_InitStructure.DAC_ConnectOnChipPeripheral      = DISABLE;
    DAC_InitStructure.DAC_ConnectExternalPin           = ENABLE;
    DAC_InitStructure.DAC_TriggerEnable                = ENABLE;
    DAC_Init(DAC1, &DAC_InitStructure);

    /* DAC2 Configuration */
    DAC_InitStructure.DAC_LFSRUnmask_TriangleAmplitude = DAC_TriangleAmplitude_1023;
    DAC_Init(DAC2, &DAC_InitStructure);

    /* Set DAC1 DAC2 DHR12R register */
    DAC_SetDualChData(DAC12,DAC_ALIGN_R_12BIT, 0x100, 0x100);
    /* Enable DAC1 */
    DAC_Enable(DAC1, ENABLE);
    /* Enable DAC2 */
    DAC_Enable(DAC2, ENABLE);
}
```

2. 用户修调可以在操作条件与标称出厂修整条件不同时进行，特别是当 VDDA 电压、温度、 VREF+值发生变化时，可以在应用程序的任何时点通过软件来进行。具体的修调请参考 “UserBufferCalibration”的 Demo 例程

4.3.30COMP

下表是N32G45x系列和N32H47x_48x系列在COMP模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用COMP模块驱动API函数可以依照下表做替换；

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
COMP 复位	void COMP_DeInit(void)	void COMP_DeInit(void)	是	无
COMP 结构体初始化	void COMP_StructInit(COMP_InitType* COMP_InitStruct)	void COMP_StructInit(COMP_InitType* COMP_InitStruct)	是	参数枚举有差异。 对比缺少了 InpDOutSelacConnect 与 OuOutSeltSel（与设计改动有关）
COMP 初始化	void COMP_Init(COMPX COMPx, COMP_InitType* COMP_InitStruct)	void COMP_Init(COMPX COMPx, COMP_InitType* COMP_InitStruct)	是	参数枚举有差异，具体参考代码
COMP 使能	void COMP_Enable(COMPX COMPx, FunctionalState en)	void COMP_Enable(COMPX COMPx, FunctionalState Cmd)	是	无
选择 COMP 正端输入	void COMP_SetInpSel(COMPX COMPx, COMP_CTRL_INPSEL VpSel)	void COMP_SetInpSel(COMPX COMPx, COMP_CTRL_INPSEL VpSel)	是	无
选择 COMP 负端输入	void COMP_SetInmSel(COMPX COMPx, COMP_CTRL_INMSEL VmSel)	void COMP_SetInmSel(COMPX COMPx, COMP_CTRL_INMSEL VmSel)	是	无
选择输出到 TIM	void COMP_SetOutTrig(COMPX COMPx, COMP_CTRL_OUTTRIG OutTrig)	void COMP_OutToTimEnable(uint32_t TimEn, FunctionalState Cmd)	否	函数改动，因为设计改动，由原来的 1 对 1 输出到 TIM，改成 1 对多输出的方式
设置 COMP 锁定	void COMP_SetLock(uint32_t Lock)	void COMP_SetLock(uint32_t Lock)	是	无
设置中断使能	void COMP_SetIntEn(uint32_t IntEn)	void COMP_SetIntEn(uint32_t IntEn, FunctionalState Cmd)	否	兼容中断的使能与失能
获取中断状态标志	uint32_t COMP_GetIntSts(void)	无	否	删除多余函数
获取某个 COMP 的中断状态	FlagStatus COMP_GetIntStsOneComp(COMPX COMPx)	FlagStatus COMP_GetIntStsOneComp(COMPX COMPx)	是	无
清除某个 COMP 的中断	无	void COMP_ClearIntStsOneComp(COMPX COMPx)	否	新增函数
设置滤波适中的分频系数	void COMP_SetFilterPrescaler(COMPX COMPx , uint16_t FilPreVal)	void COMP_SetFilterPrescaler(COMPX COMPx , uint16_t FilPreVal)	是	无
设置滤波控制	void COMP_SetFilterControl(COMPX COMPx , uint8_t FilEn, uint8_t TheresNum , uint8_t SampPW)	void COMP_SetFilterControl(COMPX COMPx , uint8_t FilEn, uint8_t TheresNum , uint8_t SampPW)	是	无
设置迟滞等级	void COMP_SetHyst(COMPX COMPx , COMP_CTRL_HYST HYST)	void COMP_SetHyst(COMPX COMPx , COMP_CTRL_HYST HYST)	是	参数枚举有差异。
设置消隐源	void COMP_SetBlanking(COMPX COMPx , COMP_CTRL_BLKING BLK)	void COMP_SetBlanking(COMPX COMPx , COMP_CTRL_BLKING BLK)	是	参数枚举有差异。
获取比较器滤波前的结果	无	FlagStatus COMP_GetOutStatus(COMPX COMPx)	是	开启滤波时，该函数返回的是滤波前的结果
设置 6bitDAC 控制输出	void COMP_SetRefScl(uint8_t Vv2Trim, bool Vv2En, uint8_t Vv1Trim, bool Vv1En)	void COMP_SetRefScl(uint8_t Vv3Trim, bool Vv3En, uint8_t Vv2Trim, bool Vv2En, uint8_t Vv1Trim, bool Vv1En)	是	无

获取比较器滤波后的结果	FlagStatus COMP_GetOutStatus(COMPX COMPx)	FlagStatus COMP_GetFilterOutStatus(COMPX COMPx);	否	新版区分滤波后与滤波前的结果
窗口比较器配置	无	void COMP_WindowModeEnable(uint32_t WinModeEn, FunctionalState Cmd)	否	新增函数
使能 VFLAG 控制	无	void COMP_SetVflagEnable(COMPX COMPx , FunctionalState Cmd);	否	新增函数

下图是N32H47x_48x系列在COMP模块的改动比较大的点，主要是新增了COMP滤波前后滤波后的状态输出，迟滞等级，COMP输出到TIM变成1对多同时输出：

1.下图的左边是G45x的输出到TIM的配置方式，右边是N32H47x_48x系列的配置方式；

```
/**
 * @brief Configures the comp module.
 */
void COMP_Configuratoin(void)
{
    COMP_InitType COMP_Initial;

    /*Set dac2,dac1. because dac1/PA4 is share pin line,so only PB0 puls 0/1, can find out puls*/
    COMP_SetRefScl(16, true, 32, true);
    /*Initial comp*/
    COMP_StructInit(&COMP_Initial);
    COMP_Initial.InpSel     = COMP1_CTRL_INPSEL_PB10;
    COMP_Initial.InmSel     = COMP1_CTRL_INMSEL_DAC1_PA4;
    COMP_Initial.SampWindow = 18;           //(0~32)
    COMP_Initial.Thresh     = 30;           //(0~32)
    COMP_Init(COMP1, &COMP_Initial);
    /*trig initial as tim1&tim8 break*/
    COMP_SetOutTrig(COMP1, COMP1_CTRL_OUTSEL_TIM1_BKIN_TIM8_BKIN);
    /*enable comp1*/
    COMP_Enable(COMP1, ENABLE);
}
```

```
/**
 *\*\name    COMP_Configuratoin.
 *\*\fun     Configures the comp module.
 *\*\return  none
 **/
void COMP_Configuratoin(void)
{
    COMP_InitType COMP_Initial;

    /*Initial comp*/
    COMP_StructInit(&COMP_Initial);
    COMP_Initial.InpSel     = COMP1_CTRL_INPSEL_PA1;
    COMP_Initial.InmSel     = COMP1_CTRL_INMSEL_PA4;
    COMP_Initial.SampWindow = 18;       //(0~32)
    COMP_Initial.Threshold  = 30;       //(0~32)
    COMP_Init(COMP1, &COMP_Initial);
    COMP_Enable(COMP1, ENABLE);
    /*Enable comp1 output to timer*/
    COMP_OutToTimEnable(COMP1_TIM_EN,ENABLE);
}
```

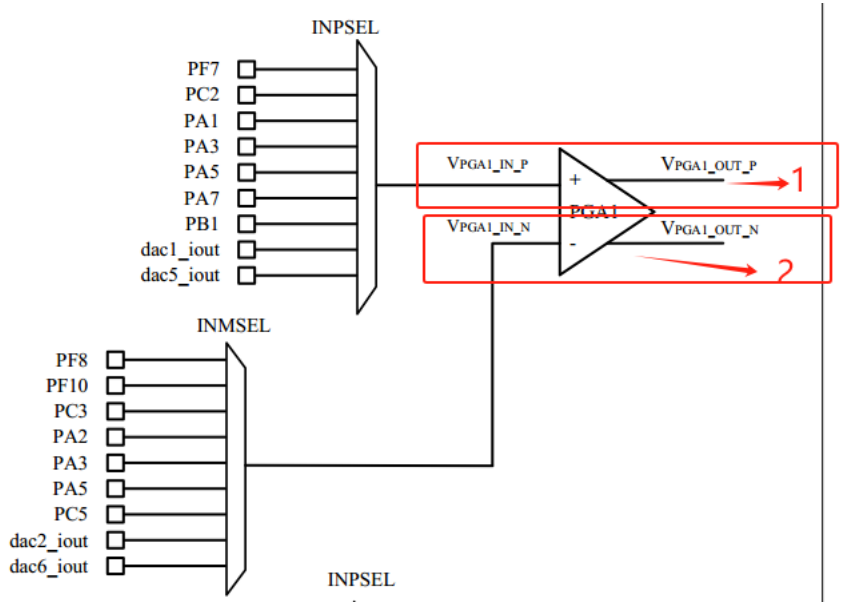
4.3.31 PGA

N32H47x_48x系列的PGA模块对比N32G45x的OPA模块的功能改动比较大，直接对比意义不大，更不能直接使用替换的方式取修改，建议参考PGA的Demo进行测试；

N32H47x_48x 系列 PGA 的输出是直接连接到 ADC 的内部通道上，不能配置连接到外部 I/O 上。PGA 可以工作在单端模式和差分模式：

4.3.31.1 单端模式：

1.以PGA1来说，PGA1可以拆分为2个单端PGA使用，INPSEL可以选择一个通道作为输入，输出直接连接的ADC1_VINP1上；

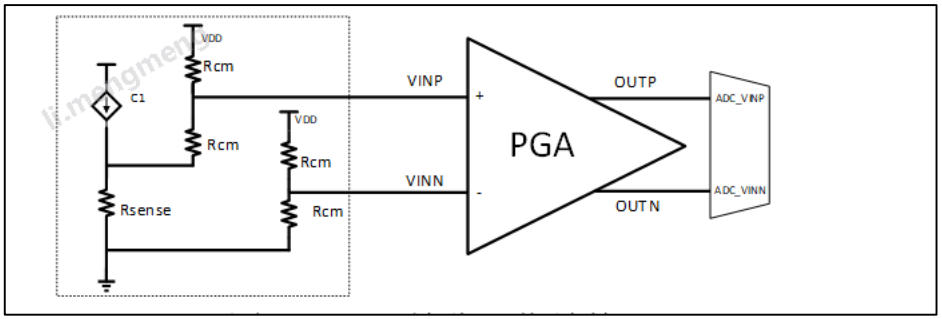


下面以PA1作为输入，2倍增益为例，对应配置代码如下图所示：

```
**\return none
**/
void PGA_Configuratoin(void)
{
    PGA_InitType PGA_Initial;

    /*Initial comp*/
    PGA_StructInit(&PGA_Initial);
    PGA_Initial.DiffModeEn    = DISABLE;
    PGA_Initial.Gain1         = PGA_CTRL_CH1GAIN_2_DIFF_4;
    PGA_Initial.Vpsel         = PGA1_CTRL_VPSEL_PA1;
    PGA_Initial.TimeAutoMuxEn  = DISABLE ;
    PGA_Initial.En1           = DISABLE;
    PGA_Initial.En2           = DISABLE;
    PGA_Init(PGA1, &PGA_Initial);
    /*enable PGA1 CH1 and CH2 */
    PGA_Enable(PGA1,PGA_Channel1, ENABLE);
}

/**
 *\*\name    ADC_Initial.
 *\*\fun     ADC_Initial program.
 *\*\return  none
 **/
void ADC_Initial(void)
{
    /* ADCx configuration ------------------------------------------------------*/
    ADC_InitStructure.WorkMode       = ADC_WORKMODE_INDEPENDENT;
    ADC_InitStructure.MultiChEn      = DISABLE;
    ADC_InitStructure.ContinueConvEn = ENABLE;
    ADC_InitStructure.ExtTrigSelect  = ADC_EXT_TRIG_REG_CONV_SOFTWARE;
    ADC_InitStructure.DatAlign       = ADC_DAT_ALIGN_R;
    ADC_InitStructure.ChsNumber      = 1;
    ADC_InitStructure.Resolution     = ADC_DATA_RES_12BIT;
    ADC_Init(ADC1, &ADC_InitStructure);

    ADC_ConfigRegularChannel(ADC1, ADC_CH_1, 1, ADC_SAMP_TIME_CYCLES_55_5);
    /* Enable ADC1 channel 1 positive end connecting to output of PGA1 positive end. */
    ADC_EnableCH1PositiveEndConnetPGA_P(ADC1,ENABLE);
    /* Enable ADC1 */
    ADC_Enable(ADC1, ENABLE);
    /*Check ADC1 Ready*/
    while(ADC_GetFlagStatus(ADC1,ADC_FLAG_RDY) == RESET)
        ;
    /* Start ADC1 single-ended calibration */
    ADC_CalibrationOperation(ADC1,ADC_CALIBRATION_SIGNAL_MODE);
    /* Check the end of ADC1 calibration */
    while (ADC_GetCalibrationStatus(ADC1))
        ;
}
```

4.3.31.2 差分模式：

1. 以 PGA1 来说，PGA1 可以组成差分 PGA 使用，INPSEL、INMSEL 可以选择一组差分通道作为输入，输出直接连接的 ADC1_VINP1 和 ADC1_VINN1 的 ADC 差分输入通道上；
2. 外部的连接图如下所示；



注意：PGA 的 VINN 需要提供固定的 VDDA/2 的电压偏置；

3，下面以PA1作为正端输入，PA2作为负端输入，2倍差分增益为例，对应配置代码如下图所示

```
void PGA_Configuratoin(void)
{
    PGA_InitType PGA_Initial;

    /*Initial comp*/
    PGA_StructInit(&PGA_Initial);
    PGA_Initial.DiffModeEn    = ENABLE;
    PGA_Initial.Gain1         = PGA_CTRL_CH1GAIN_1_DIFF_2;
    PGA_Initial.Vmsel         = PGA1_CTRL_VMSEL_PA2;
    PGA_Initial.Vpsel         = PGA1_CTRL_VPSEL_PA1;
    PGA_Initial.TimeAutoMuxEn = DISABLE ;
    PGA_Initial.En1           = DISABLE;
    PGA_Initial.En2           = DISABLE;
    PGA_Init(PGA1, &PGA_Initial);
    /*enable PGA1 CH1 and CH2 */
    PGA_Enable(PGA1,PGA_Channel1, ENABLE);
    PGA_Enable(PGA1,PGA_Channel2, ENABLE);
}

/**
*\*\name    ADC_Initial.
*\*\fun     ADC_Initial program.
*\*\return  none
**/
void ADC_Initial(void)
{
    /* ADCx configuration ------------------------------------------------------*/
    ADC_InitStructure.WorkMode       = ADC_WORKMODE_INDEPENDENT;
    ADC_InitStructure.MultiChEn      = DISABLE;
    ADC_InitStructure.ContinueConvEn = ENABLE;
    ADC_InitStructure.ExtTrigSelect  = ADC_EXT_TRIG_REG_CONV_SOFTWARE;
    ADC_InitStructure.DatAlign       = ADC_DAT_ALIGN_R;
    ADC_InitStructure.ChsNumber      = 1;
    ADC_InitStructure.Resolution     = ADC_DATA_RES_12BIT;
    ADC_Init(ADC1, &ADC_InitStructure);

    ADC_ConfigRegularChannel(ADC1, ADC_CH_1, 1, ADC_SAMP_TIME_CYCLES_55_5);
    /* Enable ADC1 channel 1 positive end connecting to output of PGA1 positive end. */
    ADC_EnableCH1PositiveEndConnetPGA_P(ADC1,ENABLE);
    /* Enable ADC1 channel 1 negtive end connecting to output of PGA1 negtive end. */
    ADC_EnableCH1NegtiveEndConnetPGA_N(ADC1,ENABLE);
    /* Enable ADC1 channel 1 work on differential mode. */
    ADC_SetChannelSingleDiff(ADC1,ADC_CH_1,ADC_DIFFERENTIAL_ENDED);
    /* Enable ADC1 */
    ADC_Enable(ADC1, ENABLE);
    /*Check ADC1 Ready*/
    while(ADC_GetFlagStatus(ADC1,ADC_FLAG_RDY) == RESET)
        ;
    /* Start ADC1 differential calibration */
    ADC_CalibrationOperation(ADC1,ADC_CALIBRATION_DIFF_MODE);
    /* Check the end of ADC1 calibration */
    while (ADC_GetCalibrationStatus(ADC1))
        ;
}
/**
```

4.3.31.3 用户TRIM：

PGA 由于放大器的特征，当校准环境与用户的工作环境差异较大时，为了补偿各器件带来的误差，支持用户修调来校准PGA，具体参考“PGA_UserTrimming”示例工程；

4.3.32VREFBUF

N32H47x_48x系列的VREFBUF模块属于新增功能，建议参考VREFBUF的Demo进行测试；

注意：在某些封装中，如果将 VREF+引脚与 VDDA 引脚连接到一起时（具体可以参考数据手册的封装引脚描述），电压参考缓冲器不可用，必须保持禁用且需要将 RCC_VREFCTRL.HIM 位置 1：

下图是N32H47x_48x系列VREFBUF模块的“ADC_VREFBUF”示例工程的部分配置代码：

```
/**
*\*\name    VREFBUF_CONFIG.
*\*\fun     Configures vrefbuf work mode.
*\*\param   VrefBufferMode.
*\*\          - VREFBUF_MODE_OUTPUT
*\*\          - VREFBUF_MODE_INPUT:
*\*\param   voltage
*\*\          - VREFBUF_VOLTAGE_SCALE_2_048V
*\*\          - VREFBUF_VOLTAGE_SCALE_2_5V
*\*\          - VREFBUF_VOLTAGE_SCALE_2_9V
*\*\return  none
**/
void VREFBUF_CONFIG(VREFBUF_MODE VrefBufferMode, uint32_t voltage)
{

    if(VrefBufferMode == VREFBUF_MODE_OUTPUT)
    {
        *(uint32_t *)VREFBUF_EN_CTRL |= (1<<10);
        VREFBUF_SetVoltageScale(voltage);
        VREFBUF_EnableHIM(DISABLE);
        VREFBUF_Enable(ENABLE);
        while(VREFBUF_IsVREFReady() == RESET);
    }
    else
    {
        *(uint32_t *)VREFBUF_EN_CTRL &= (~(1<<10));
        VREFBUF_EnableHIM(ENABLE);
        VREFBUF_Enable(DISABLE);
    }
}
/**
```

注意：使用VREFBUF输出时， VREF+引脚不能外接电压

4.3.33 XSPI

下表是N32G45x系列和N32H47x_48x系列在XSPI模块的驱动API函数对比表，替换驱动.c/.h文件后，应用代码调用XSPI模块驱动API函数可以依照下表做替换：

API 描述	API 名称		API 是否一致	差异说明
	N32G45x 系列	N32H47x_48x 系列		
XSPI 使能	void QSPI_Cmd(bool cmd)	void XSPI_Cmd(FunctionalState cmd)	否	函数名不一致
XIP 使能	void QSPI_XIP_Cmd(bool cmd)	void XSPI_XIP_Cmd(FunctionalState cmd)	否	函数名不一致

XSPI 复位	void QSPI_DeInit(void)	void XSPI_DeInit(void)	否	函数名不一致
XSPI 初始化	void QspiInitConfig(QSPI_InitType* QSPI_InitStruct)	void XSPI_Init(XSPI_InitType* XSPI_InitStruct)	否	函数名及输入参数结构体不一致：QSPI_InitStruct→XSPI_InitStruct
XSPI 结构体初始化	-	void XSPI_StructInit(XSPI_InitType* XSPI_InitStruct)	否	N32G45x 系列无此 API
引脚配置	void QSPI_GPIO(QSPI_NSS_PORT_SEL qspi_nss_port_sel, bool IO1_Input, bool IO3_Output)	-	否	N32H47x_48x 系列无此 API
DMA 发送阈值及使能	void QSPI_Tx_DMA_CTRL_Config(uint8_t Cmd,uint8_t TxDataLevel)	void XSPI_ConfigDMATxLevel(uint32_t TxDataLevel)	否	N32H47x_48x 系列将 DMA 发送阈值和使能拆分为两个函数，新增 XSPI_DMAREQ_TX 参数
		void XSPI_EnableDMA(XSPI_DMAREQ_TX, FunctionalState Cmd)	否	
DMA 接收阈值及使能	void QSPI_Rx_DMA_CTRL_Config(uint8_t Cmd, uint8_t RxDataLevel)	void XSPI_ConfigDMARxLevel(uint32_t RxDataLevel)	否	N32H47x_48x 系列将 DMA 接收阈值和使能拆分为两个函数，新增 XSPI_DMAREQ_RX 参数
		void XSPI_EnableDMA(XSPI_DMAREQ_RX, FunctionalState Cmd)	否	
中断配置	-	void XSPI_ConfigInt(uint16_t XSPI_IT, FunctionalState Cmd)	否	N32G45x 系列无此 API
中断状态获取	uint16_t QSPI_GetITStatus(uint16_t FLAG)	uint16_t XSPI_GetINTStatus(uint16_t FLAG)	否	函数名及输入参数 FLAG 不一致： N32G45x 系列参数： QSPI_ISTS_TXFEIS QSPI_ISTS_TXFOIS QSPI_ISTS_RXFUIS QSPI_ISTS_RXFOIS QSPI_ISTS_RXFFIS QSPI_ISTS_MMCIS QSPI_ISTS_XRXOIS N32H47x_48x 系列参数： XSPI_ISTS_TXFEIS XSPI_ISTS_TXFOIS XSPI_ISTS_RXFUIS XSPI_ISTS_RXFOIS XSPI_ISTS_RXFFIS XSPI_ISTS_MMCIS XSPI_ISTS_XRXOIS XSPI_ISTS_TXUIS

中断状态清除	void QSPI_ClearITFLAG(uint16_t FLAG)	void XSPI_ClearITBit(uint16_t XSPI_IT)	否	函数名及输入参数 FLAG 不一致： N32G45x 系列参数： QSPI_ISTS_TXFEIS QSPI_ISTS_TXFOIS QSPI_ISTS_RXFUIS QSPI_ISTS_RXFOIS QSPI_ISTS_RXFFIS QSPI_ISTS_MMCIS QSPI_ISTS_XRXOIS QSPI_XIP_RXFOI_CLR_XRXFOIC N32H47x_48x 系列参数： XSPI_IT_TXUIM XSPI_IT_XRXOIM XSPI_IT_MMCIM XSPI_IT_RXFFIM XSPI_IT_RXFOIM XSPI_IT_RXFUIM XSPI_IT_TXFOIM XSPI_IT_TXFEIM
	void QSPI_XIP_ClearITFLAG(uint16_t FLAG)		否	
标志获取	-	FlagStatus XSPI_GetFlagStatus(uint32_t XSPI_FLAG)	否	N32G45x 系列无此 API
获取 busy 状态	bool GetQspiBusyStatus(void)	FlagStatus XSPI_GetBusyStatus(void)	否	函数命名不一致
获取发送 FIFO 满状态	bool GetQspiTxDataBusyStatus(void)	FlagStatus XSPI_GetTxDataBusyStatus(void)	否	函数命名不一致
获取发送 FIFO 空状态	bool GetQspiTxDataEmptyStatus(void)	FlagStatus XSPI_GetTxDataEmptyStatus(void)	否	函数命名不一致
获取接收 FIFO 空状态	bool GetQspiRxHaveDataStatus(void)	FlagStatus XSPI_GetRxHaveDataStatus(void)	否	函数命名不一致
获取接收 FIFO 满状态	bool GetQspiRxDataFullStatus(void)	FlagStatus XSPI_GetRxDataFullStatus(void)	否	函数命名不一致
获取数据冲突状态	bool GetQspiDataConflictErrorStatus(void)	FlagStatus XSPI_GetDataConflictErrorStatus(void)	否	函数命名不一致
发送 1 个字数据	void QspiSendWord(uint32_t SendData)	void XSPI_SendData(uint32_t SendData)	否	函数命名不一致
读取 1 个字数据	uint32_t QspiReadWord(void)	uint32_t XSPI_ReceiveData(void)	否	函数命名不一致
获取 DAT0 寄存器指针	uint32_t QspiGetDataPointer(void)	uint32_t XSPI_GetDataPointer(void)	否	函数命名不一致
获取接收 FIFO 的数据个数	uint32_t QspiReadRxFifoNum(void)	uint32_t XSPI_ReadRxFifoNum(void)	否	函数命名不一致

获取发送 FIFO 的数据个数	-	uint32_t XSPI_ReadTxFifoNum(void)	否	N32G45x 系列无此 API
清除接收 FIFO	void ClrFifo(void)	void XSPI_ClrFifo(void)	否	函数命名不一致
读取 N 个字数据	uint32_t GetFifoData(uint32_t* pData, uint32_t Len)	uint32_t XSPI_GetFifoData(uint32_t* pData, uint32_t Len)	否	函数命名不一致
发送 N 个字，并获取返回的数据	void QspiSendAndGetWords(uint32_t* pSrcData, uint32_t* pDstData, uint32_t cnt)	void XSPI_SendAndGetWords(uint32_t* pSrcData, uint32_t* pDstData, uint32_t cnt)	否	函数命名不一致
发送 1 个字的数据，并获取返回的数据	uint32_t QspiSendWordAndGetWords(uint32_t WrData, uint32_t* pRdData, uint8_t LastRd)	uint32_t XSPI_SendWordAndGetWords(uint32_t WrData, uint32_t* pRdData, uint8_t LastRd)	否	函数命名不一致
设置传输类型	-	void XSPI_SetTransType(uint32_t TransType)	否	N32G45x 系列无此 API
设置等待周期	-	void XSPI_SetWaitCycles(uint32_t WAITCYCLES)	否	N32G45x 系列无此 API
设置接收 FIFO 满阈值	-	void XSPI_SetRXFIFOLevel(uint32_t fifo_len)	否	N32G45x 系列无此 API
设置发送 FIFO 空阈值	-	void XSPI_SetTXFIFOLevel(uint32_t fifo_len)	否	N32G45x 系列无此 API
设置发送 FIFO 开始传输的阈值	-	void XSPI_SetTXFIFOStartLevel(uint32_t fifo_len)	否	N32G45x 系列无此 API
获取接收 FIFO 满阈值	-	uint8_t XSPI_GetRXFIFOLevel(void)	否	N32G45x 系列无此 API
获取发送 FIFO 空阈值	-	uint8_t XSPI_GetTXFIFOLevel(void)	否	N32G45x 系列无此 API

N32G45x系列和N32H47x_48x系列在XSPI模块的示例工程代码差异较大，建议直接参考如下N32H47x_48x系列例程，根据代码及readme完成应用开发：


projects
n32h47x_48x_EVAL
examples
XSPI
名称
修改日期
XSPI_DUALQUAD
XSPI_OCTAL
XSPI_QUAD
XSPI_QUAD_DMA
2024/4/29 10:29
2024/4/29 10:29
2024/4/29 10:29
2024/4/29 10:29
双四线模式
八线模式
四线模式
四线+DMA模式

5 版本历史

日期	版本	备注
2024.05.09	V0.1.0	初始版本
2024.11.12	V1.0.0	新增资源对比及硬件差异

6 声明

国民技术股份有限公司（下称“国民技术”）对此文档拥有专属产权。依据中华人民共和国的法律、条约以及世界其他法域相适用的管辖，此文档及其中描述的国民技术产品（下称“产品”）为公司所有。
国民技术在此并未授予专利权、著作权、商标权或其他任何知识产权许可。所提到或引用的第三方名称或品牌（如有）仅用作区别之目的。
国民技术保留随时变更、订正、增强、修改和改良此文档的权利，恕不另行通知。请使用人在下单购买前联系国民技术获取此文档的最新版本。
国民技术竭力提供准确可信的资讯，但即便如此，并不推定国民技术对此文档准确性和可靠性承担责任。
使用此文档信息以及生成产品时，使用者应当进行合理的设计、编程并测试其功能性和安全性，国民技术不对任何因使用此文档或本产品而产生的任何直接、间接、意外、特殊、惩罚性或衍生性损害结果承担责任。
国民技术对于产品在系统或设备中的应用效果没有任何故意或保证，如有任何应用在其发生操作不当或故障情况下，有可能致使人员伤亡、人身伤害或严重财产损失，则此类应用被视为“不安全使用”。
不安全使用包括但不限于：外科手术设备、原子能控制仪器、飞机或宇宙飞船仪器、所有类型的安全装置以及其他旨在支持或维持生命的应用。
所有不安全使用的风险应由使用人承担，同时使用人应使国民技术免于因为这类不安全使用而导致被诉、支付费用、发生损害或承担责任时的赔偿。
对于此文档和产品的任何明示、默示之保证，包括但不限于适销性、特定用途适用性和不侵权的保证责任，国民技术可在法律允许范围内进行免责。
未经明确许可，任何人不得以任何理由对此文档的全部或部分进行使用、复制、修改、抄录和传播。