沉芯异构芯片

V 1.5

若贝（无锡）微电子有限公司

Robei 沉芯异构芯片不同于传统的 CPU/GPU，是一种采用串行计算的 CPU 加上并行计算的动态可重构阵列（CGRA 架构）组成的异构系统芯片，该芯片具备控制流的串行计算和数据流的并行计算，并且可以在串并行计算之间自由切换，一颗芯片可以替代 CPU+DSP、CPU+FPGA 或 CPU+DSP+FPGA 的多芯片系统。沉芯异构芯片支持开放指令集 RISC-V，自主设计整体架构，尤其是动态可重构的自适应计算阵列获得了中国、美国、加拿大的发明专利，拥有相应的软硬件开发平台，为用户提供快速便捷的开发环境。

本资料产品所针对的应用领域为：嵌入式、数字信号处理、机器人、智能家电、智能家居、物联网、智能玩具、电机控制等。

沉芯®

目录

1.沉芯异构芯片

1.1 沉芯系列概述

沉芯系列芯片，支持 RISC-V 开放指令集的 RV32IM，三级流水，支持 10MHZ~200MHZ 主频， 0~192 个 32 位元的动态可重构计算单元、0~26 个 32 位乘法器及 FP32 计算单元（IEEE-754 标准）、针对 32 位数据的最大计算能力 38GOPS、通用接口为 SPI、QSPI、UART、PWM、I2C、SDRAM（RAC101 和 RAC102 不含）、按组可配置 GPIO 等。

1.2 产品选型列表

为了便于客户进行选择，表 1-1 具体列出了不同型号的配置列表。其中，RAC101 系列不具备动态可重构计算单元，可以作为一个独立的 MCU 运行。

表 1-1 选型参数对比

类别	型号	CPU	乘除法器	频率 MHZ	可重构单元	浮点计算单元数	最大算力 GOPS	SRAM 数据存储	SRAM 指令存储	I2C	SPI/QSPI	UART	PWM	可编程 GPIO
沉芯	RAC101	RV32IM	Yes	10~100	0	0	0.1	16	24	2	2	2	8	32
	RAC102	RV32IM	Yes	10~200	72	12	12.8	16	24	2	2	2	16	32
	RAC103	RV32IM	Yes	10~200	192	26	38.4	32	24	2	2	2	20	80

表 1-2 命名规则

芯片系列	芯片类别	温度等级	封装形式	引脚数目	额外功能
RAC	102	I	Q	064	X

1. 芯片系列：RAC 是 Robei Adaptive Chip 的缩写
2. 芯片类别：1xx 系列是沉芯系列芯片的编号：
   101 是纯 MCU；
   102 是 MCU+72 核动态可重构；
   103 是 MCU+192 核动态可重构；
3. 温度等级：I 代表-40~85 度；
   C 代表 0~70 度；
   R 代表-40~125 度；
4. 封装形式：Q 代表 QFN 封装。L 代表 LQFP 封装，B 代表 BGA 封装。
5. 引脚数目：三位数代表着引脚数目。064 是 64 个引脚。
6. 额外功能：N 是不带额外的功能，F 是带内置 Flash，R 是带内部的 RTC。（此

处根据客户需求，可以订制功能性 SIP 封装）。

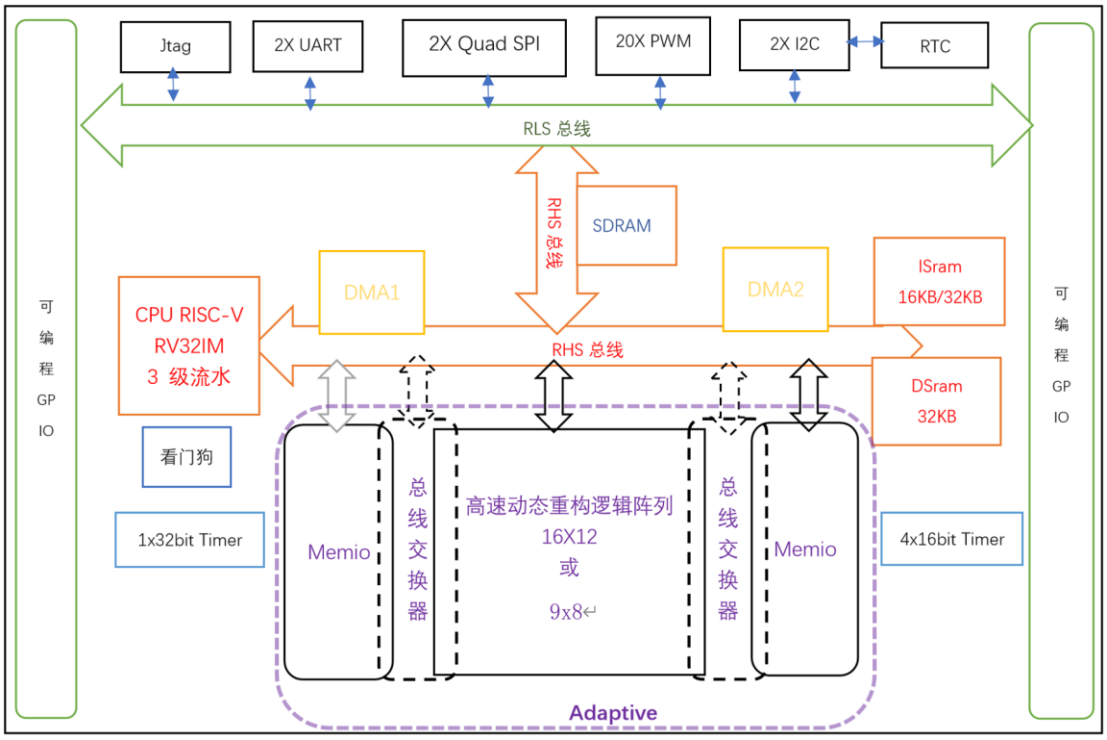


图 1-1 沉芯芯片结构示意图

1.3 Robei 沉芯®系列

并行计算

- 0~192 个并行可变逻辑核心
- 每个计算单元都是 32 位元
- 0~26 个浮点计算单元
- 16KB~32KB 数据缓存
- 24KB 指令缓存
- 最大计算能力 0.1~38.4GOPS
- 支持向量、矩阵、数据流处理
- 支持图像卷积运算、插值运算等

CPU

- RISC-V 指令集：RV32IM
- CPU 频率 10MHZ~200MHZ
- CPU 总线位宽 32bit
- 支持操作系统 RTOS
- 支持编译器 GCC
- 可用 C/C++编程
- 内核 1.2V，外部 3.3V 电压
- 快速中断响应处理
- 指令 SRAM：24KB
- 数据 SRAM：16KB～32KB

动态重构

- 指令配置方式，最少 16 条指令
- 重构一次需要几微秒
- 计算单元 10 向连接
- 部分重构不需要重启
- 可暂停数据重配置再传输数据

逻辑门数

- RAC101 有 420 万逻辑门
- RAC102 有 840 万逻辑门
- RAC103 预估 1520 万逻辑门

I/O 管脚

- 32~80 个按组可编程 GPIO
- 2 个 SPI/Quad SPI 接口

- 2 个 UART 接口
- 8-20 个 PWM 接口
- 2 个 I2C 接口

时钟

- 10MHZ~200MHZ 运行频率
- 可变逻辑内部可以降频

低功耗

- 减少取指、译码、取值等操作
- 降低对存储的读写操作
- 每个可变逻辑单元将值传给下一级直接计算
- 不用的计算单元可以将时钟关断
- 可降低内部的计算时钟频率，降低功耗
- 降低对外设的反复读写操作

产品等级

- I：工业级
- C：民用级

产品型号

- RAC101I/C
- RAC102I/C
- RAC103I/C

1.4 动态可重构阵列

芯片设计行业自从遵循片上系统（SOC）的设计理念后，在单颗芯片上要求集成功能越来越多，在维持原芯片面积不变的情况下，需要更先进工艺来支撑复杂的功能集合体。但是很多功能并不是同时在用，造成了漏电功耗虚增和资源浪费。Robei 沉芯芯片采用时间换空间的做法，不需要很大的硅片面积，制成工艺也不一定那么先进，只要切换速度够快（几纳秒到几微秒），就可以分时复用，如同数模转换一样，数字采样频率足够高，模拟信号就可以被近似地还原回来，同样道理，只要切换地速度够快，单一小的 Robei 芯片，可以做成不同功用的加速器，实现用户体验的自适应。

自适应单元在运行中不需要反复操作指令和数据存储器，只要数据进来，出去就是结果。自适应芯片架构融合了 CGRA 和 MCU 架构，采用邻域相连的异构计算单元架构，数据走向和计算均可以通过配置数据配置，配置完后就可以变成 FPGA 的运行模式，但是与 FPGA 比，时序可控而且灵活。每次配置均是一种高并行的计算数据流图。不同于 GPU 的分块本地化处理，自适应芯片允许数据交叉，可高速擦除。芯片重构一次的时间需要几微秒（10^-6 秒），如果是部分重构，可以控制在纳秒级别。

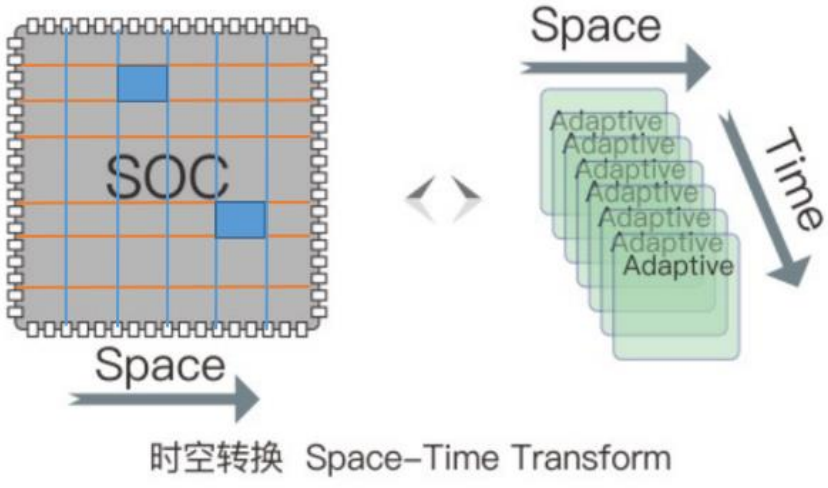


图 1-2 时空转换（Patent by US、Canada、China, etc.）

MCU 架构是非常优秀的控制器，可以负责外设的控制与调度，但是在计算能力上明显处于弱势。DSP 芯片在信号处理计算上具备非常大的优势，越来越多的项目需求已经脱离了单一芯片能完成的任务，所以很多工程项目在寻求控制与计算的一个融合架构。沉芯芯片就是一颗动态可重构的 MCU（RISC-V 指令集）融合了动态可重构的自适应架构的异构芯片，合二为一，打造出更低成本、更高速的控制与计算一体化的动态可重构芯片。

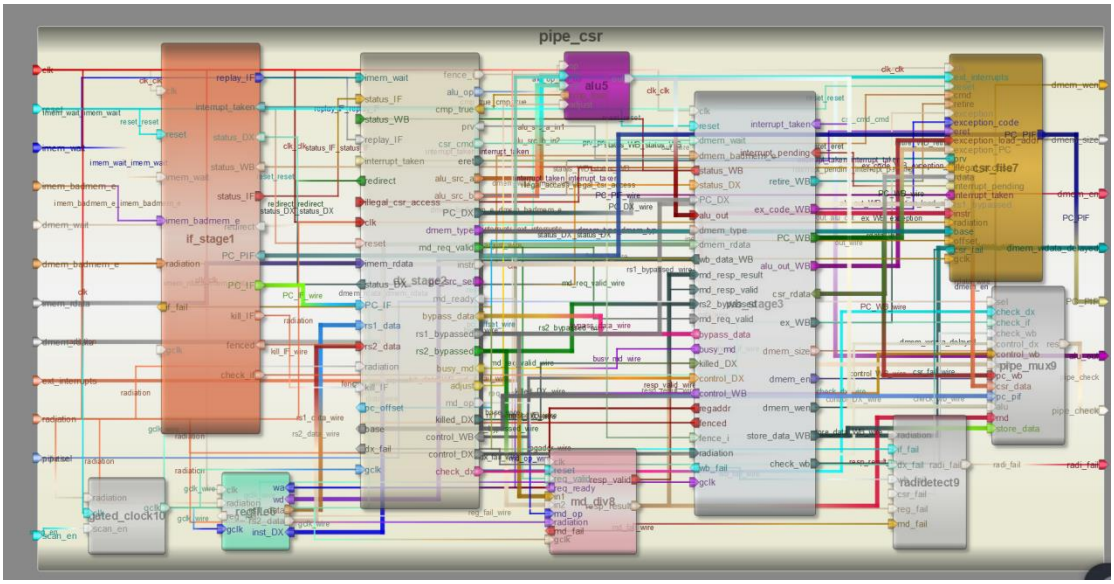


图 1-3 基于 Robei EDA 工具自主研发的 RISC-V 三级流水 CPU

沉芯系列芯片是 Robei 芯片中面向嵌入式应用开发的一款特定芯片，其内核采用了自主设计的支持 RISC-V 指令集（RV32IM）的三级流水架构，设计运行频率高达 200MHZ，具备 2 个 DMA。运行内核电压 1.2V，IO 电压 3.3V。内置 24K 指令 SRAM，16 到 32K 数据 SRAM。根据型号的划分不同，设计的动态可重构逻辑阵列的数目不同。RAC101 不带有动态可重构阵列，RAC102 带有 9 行 8 列动态可重构单元，RAC103 带有 16 行 12 列动态可重构单元。

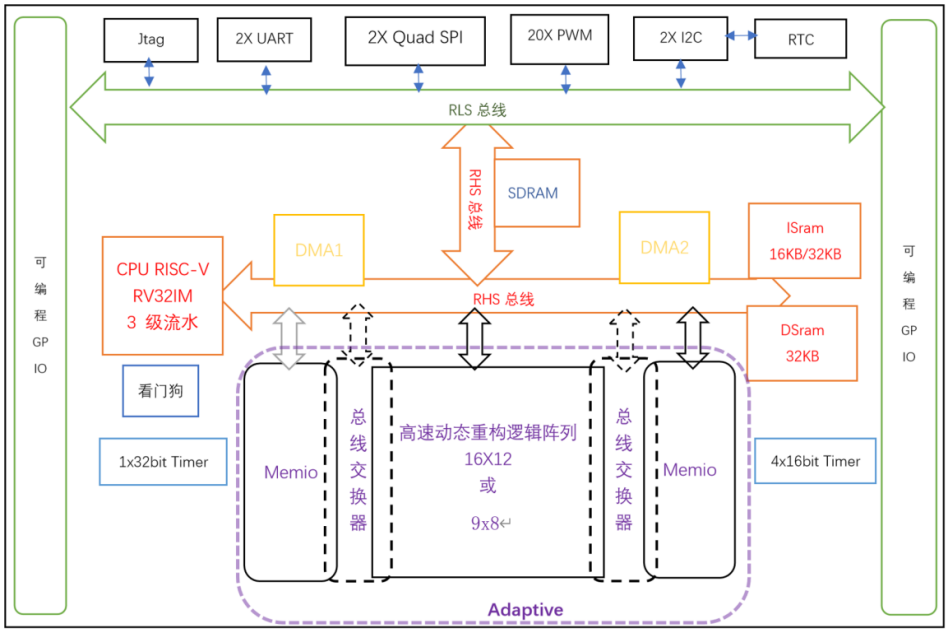


图 1-4 沉芯芯片架构

1.4.1 RAC101IQ064N 和 RAC102IQ064N

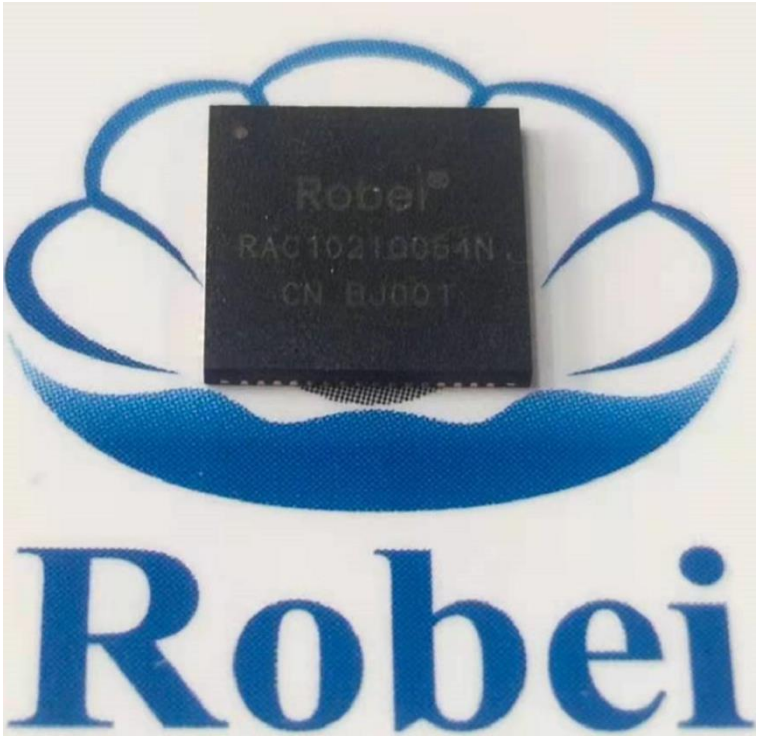


图 1-5 沉芯芯片 QFN64 封装

备注：封装形式可根据客户需求订制，亦可提供裸带。

表 1-3 在没有启动引脚映射情况下的物理引脚排布

引脚编号	功能名称	备注

1	HOLD	接 Flash 的 HOLD
2	WP	接 Flash 的 WP
3	MDI	接 Flash 的 DI
4	MDO	接 Flash 的 DO
5	MCS	接 Flash 的 CS 端
6	MCLK	接 Flash 的 CLK 端
7	A7	GPA7
8	A6	GPA6
9	A5	GPA5
10	A4	GPA4
11	VCC	3.3V 电源
12	A3	GPA3
13	A2	GPA2
14	A1	GPA1
15	A0	GPA0
16	VDD	1.2V 电源
17	B0	GPB0
18	B1	GPB1
19	B2	GPB2
20	B3	GPB3
21	VCC	3.3V 电源
22	B4	GPB4
23	B5	GPB5
24	B6	GPB6
25	B7	GPB7
26	F0	GPF0
27	F1	GPF1
28	F2	GPF2
29	F3	GPF3
30	VDD	1.2V 电源
31	F4	当 PRG 为 1 时，作为 PC 端的 FLASH 烧写的 DI； 当 PRG 为 0 时，为 F4
32	F5	当 PRG 为 1 时，PC 端的 FLASH 烧写的 CLK； 当 PRG 为 0 时，为 F5
33	C0	GPC0
34	C1	GPC1
35	C2	GPC2
36	C3	GPC3
37	HRC*	高可靠配置，设置为高电平则启动，低电平则不启动。
38	C4	GPC4
39	C5	GPC5
40	C6	GPC6

41	C7	GPC7
42	JCLK	接 JTAG 的 TCLK
43	GND	接地
44	JTDO	接 JTAG 的 TDO
45	JTMS	接 JTAG 的 TMS
46	JTDI	接 JTAG 的 TDI
47	FIQ	FIQ 中断触发，下降沿触发
48	GND	接地
49	OSCIN	接有源晶振 OUT，系统时钟
50	PRG	Flash 烧写，当 PRG=1 的时候，F4~F7 用于烧写 Flash；当 PRG=0 的时候，F4~F7 用于普通 IO。
51	RST	Reset，低电平有效
52	RCLK	RTC 的时钟，接 32.768 有源晶振
53	F6	当 PRG 为 1 时，PC 端的 FLASH 烧写的 D0； 当 PRG 为 0 时，为 F6
54	F7	当 PRG 为 1 时，PC 端的 FLASH 烧写的 CS，此时 F7 与地之间要接一个 LED 指示灯； 当 PRG 为 0 时，为 F7
55	D7	GPD7
56	D6	GPD6
57	D5	GPD5
58	D4	GPD4
59	VCC	3.3V 电源
60	D3	GPD3
61	D2	GPD2
62	D1	GPD1
63	D0	GPD0
64	VDD	1.2V 电源

沉芯芯片支持通过软件对内部虚拟引脚按照组进行重构（Remap），每个组八个引脚，可以和物理引脚进行分组映射，比如代表虚拟引脚的 GPA，可以映射到物理的引脚 C7~C0，代表虚拟引脚的 GPD,可以映射到物理引脚 B7~B0。每组映射比特位也相互对应，如 GPA0 对应 C0， GPA5 对应 C5 等。

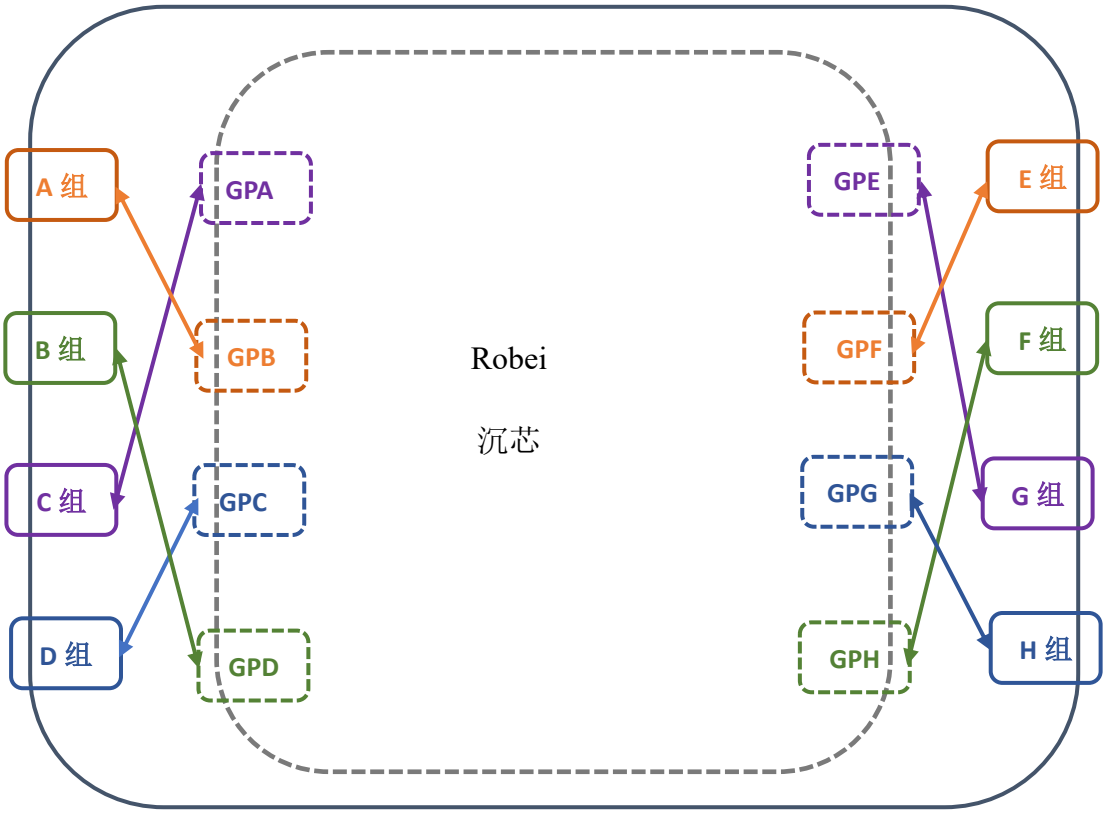


图 1-6 沉芯芯片内部虚拟引脚与物理引脚按组重构（Remap）

表 1-4 虚拟引脚功能分布

虚拟引脚组名	编号	支持重构功能
GPA	0	UART0_TXD
	1	UART0_RXD，TM32_PWM0
	2	UART1_RXD，TM32_nPWM0，TM16_1_PWM0
	3	UART1_TXD， TM16_1_nPWM0
	4	I2C0_SDA
	5	I2C0_SCL
	6	I2C1_SDA，TM32_PWM1
	7	I2C1_SCL，TM32_nPWM1
GPB	0	TM16_0_PWM0
	1	TM16_0_nPWM0
	2	TM16_0_PWM1
	3	TM16_0_nPWM1
	4	TM16_0_PWM2
	5	TM16_0_nPWM2
	6	TM16_0_PWM3
	7	TM16_0_nPWM3

GPC	0	
	1	SPI0_DO
	2	SPI0_DI
	3	SPI0_WP
	4	SPI0_HOLD
	5	SPI0_CS
	6	SPI0_CLK
	7	
GPD	0	TM32_PWM0
	1	TM32_nPWM0
	2	TM32_PWM1
	3	TM32_nPWM1
	4	TM32_PWM2
	5	TM32_nPWM2
	6	TM32_PWM3
	7	TM32_nPWM3
GPE	0	SPI1_DO，TM32_PWM2
	1	SPI1_DI，TM32_nPWM2
	2	SPI1_WP
	3	SPI1_HOLD
	4	
	5	
	6	SPI1_CS
	7	SPI1_CLK，TM16_0_nPWM3
GPF	0	TM16_1_PWM0
	1	TM16_1_nPWM0
	2	TM16_1_PWM1
	3	TM32_PWM0
	4	TM32_PWM1
	5	TM32_nPWM0
	6	TM32_nPWM1
	7	TM16_1_nPWM1
GPG	0	TM16_1_PWM2
	1	TM16_1_nPWM2
	2	TM16_1_PWM3
	3	TM16_1_nPWM3
	4	TM16_2_PWM0
	5	TM16_2_nPWM0
	6	TM16_2_PWM1
	7	TM16_2_nPWM1
GPH	0	TM16_2_PWM2
	1	TM16_2_nPWM2
	2	TM16_2_PWM3

	3	TM16_2_nPWM3
	4	TM16_3_PWM0
	5	TM16_3_nPWM0
	6	TM16_3_PWM1
	7	TM16_3_nPWM1
GPI	0	TM16_3_PWM2
	1	TM16_3_nPWM2
	2	TM16_3_PWM3
	3	TM16_3_nPWM3
	4	TM16_0_Break
	5	TM16_1_Break
	6	TM16_2_Break
	7	TM16_3_Break

注意：分组映射可以将 GPIO 内部虚拟引脚和物理引脚建立起不同的对应关系，实现内部逻辑的分时复用，但是要求每个组内对应的引脚编号不能变化。比如，GPA 中第 0 位为 UART_TXD, 第 4 位为 I2C0_SDA，如果映射到物理引脚 C 上，则 C0 需要接 UART 的 TX 引脚，第 4 位需要接 I2C 的 SDA 引脚。

1.5 电气特性

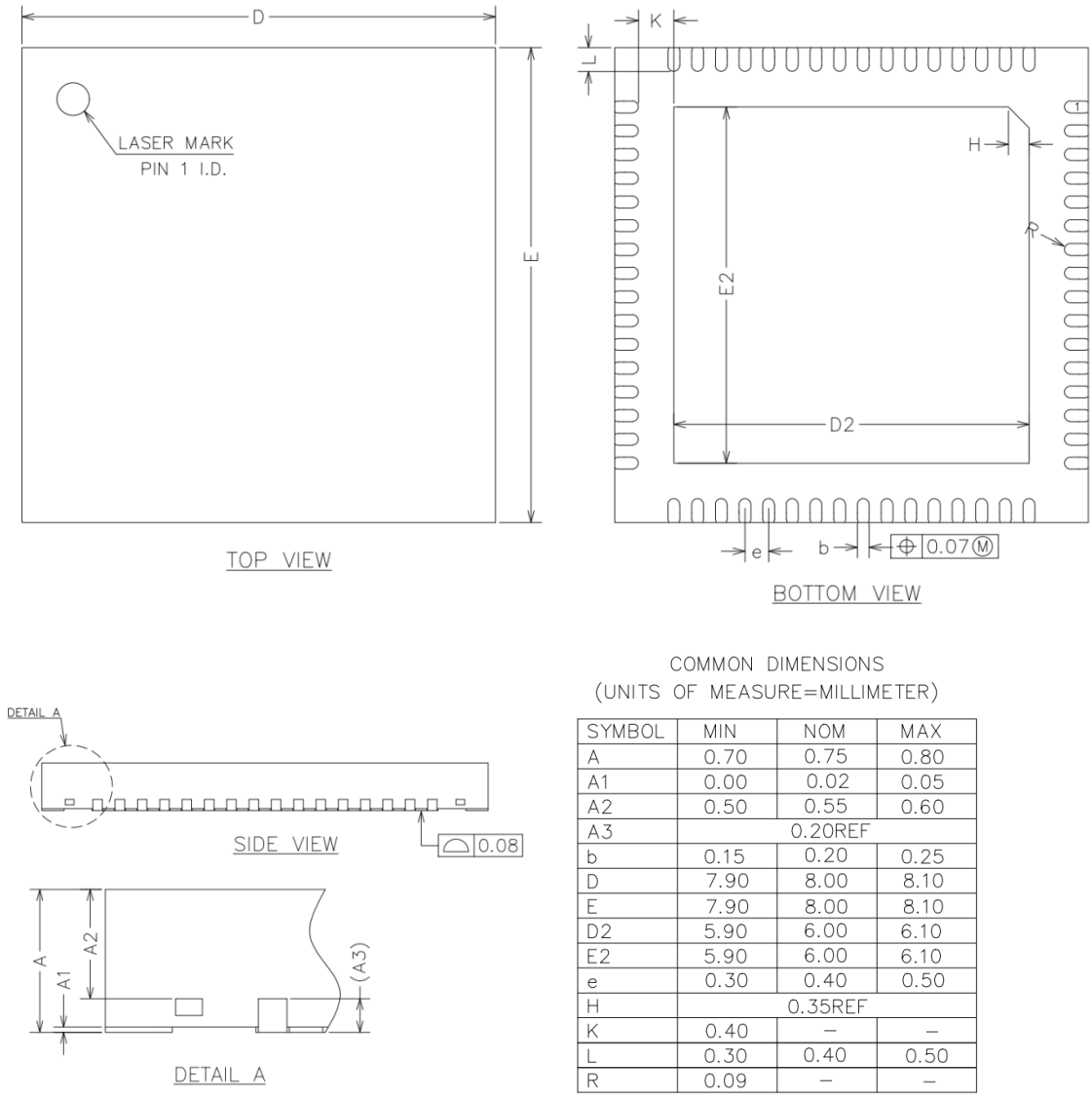
表 1-5 绝对最大额定值

符号	参数	最小值	常规值	最大值	单位
Vdd	内核电压	1.08V	1.2	1.32V	V
Vcc	引脚 IO 电压	2.97	3.3	3.63	V
VIH	高输入电压	1.7		5.5	V
VIL	低输入电压	-0.3		0.7	V
IIO	GPIO 最大电流	--	--	24	mA
Topt	工作温度	-40	--	125	℃
Tstg	储存温度	-65	--	150	℃
TJ	结温	0	25	125	℃

最大额定值仅代表在最大额定值限定范围内，芯片不会出现永久性损坏，不代表芯片可以在最大额定值时运行正常。如果长时间处于绝对最大额定值，芯片的可靠性会受到影响。

1.6 封装形式

1.6.1 QFN64 封装



COMMON DIMENSIONS
(UNITS OF MEASURE=MILLIMETER)

SYMBOL	MIN	NOM	MAX
A	0.70	0.75	0.80
A1	0.00	0.02	0.05
A2	0.50	0.55	0.60
A3		0.20REF	
b	0.15	0.20	0.25
D	7.90	8.00	8.10
E	7.90	8.00	8.10
D2	5.90	6.00	6.10
E2	5.90	6.00	6.10
e	0.30	0.40	0.50
H		0.35REF	
K	0.40	–	–
L	0.30	0.40	0.50
R	0.09	–	–

图 1-7 QFN64 封装尺寸图

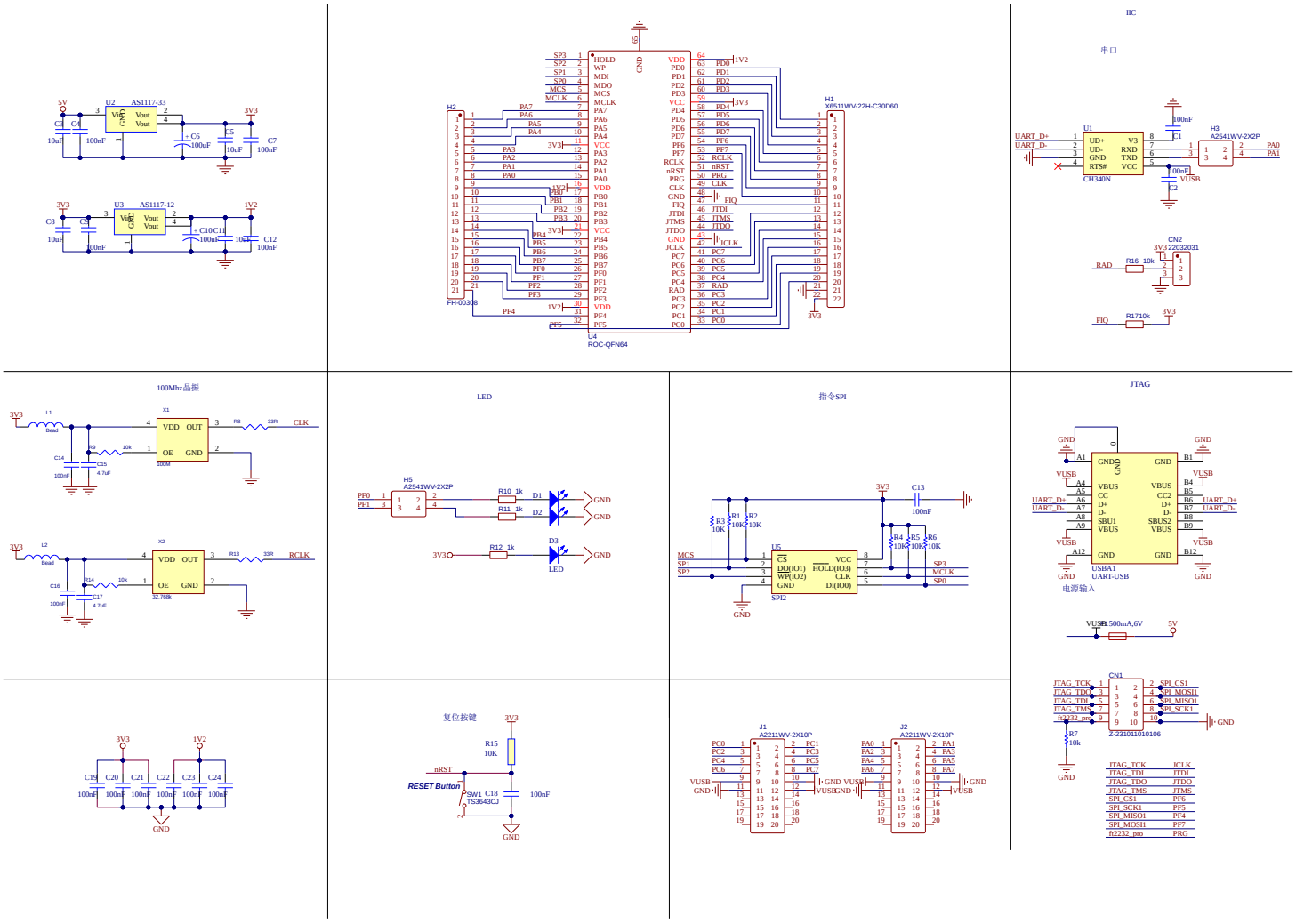
（如果客户需求量比较大，可以订制不同的封装形式：QFN64，QFN100，SIP 封装等）

1.7 参考设计

参考电路设计的原理图如图 1-8 所示，外围电路的有源晶振可以根据用户的需要选择不同频率的晶振，Vcc 是 IO 引脚上的电压（3.3V），Vdd 是内核电压

（1.2V）。

图 1-8 参考原理图设计



1.8 联系方式

若贝（无锡）微电子有限公司
地址：中国江苏省无锡市滨湖区建筑路 777 号 A10 楼 206
电话：18816653091
邮箱: **robei@robei.com**
网址: **http://robei.com**

2.Robei IDE 工具

2.1 Robei IDE 结构介绍

1. 双击桌面图标Robei或可执行文件 Robei.exe 打开 Robei IDE 软件，在菜单栏中选择 File—>New Project 创建新的项目，如图 2-1 所示。

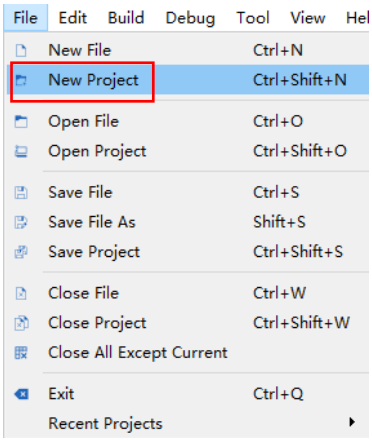


图 1-1 新建项目

2. 如图 3 所示，在弹出的窗口中设置如下参数：

- Project Name：项目命名。
- Project Path：项目路径，项目路径不允许有中文和空格。

然后点击 OK。

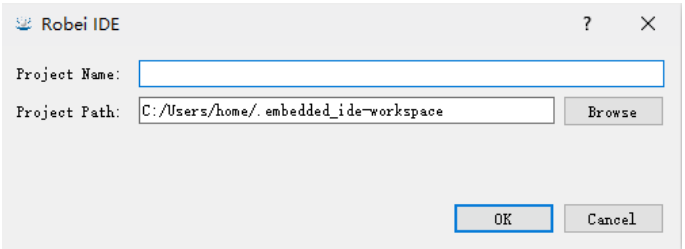


图 2-2 设置项目名和路径

3. 创建新的工程后，可以点击 File—>New File 创建新的文件，如图 2-3 所示。在弹出的界面输入文件名字和保存类型，然后点击保存。

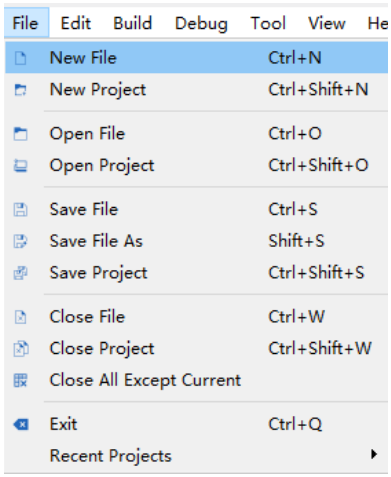


图 2-3 创建新文件

4. 点击菜单栏的 Edit 可进行编辑，打开后共有 14 项内容，如图 2-4 所示。

- Undo 恢复，选择这一项可以撤销上一步操作；
- Redo 重做，选择这一项可以恢复使用 Undo 撤销的操作；
- Cut、Copy、Paste、Delete 可实现剪切、复制、粘贴、删除操作；
- Go Next、Go Previous 跳到下/上一个编辑器标签；
- Find and Replace 在当前标签页中进行查找和替换；
- Find in Project 通过项目路径查找；
- Jump to Line 跳转到某一行代码，快速定位；
- Fold Code 折叠代码；
- Comment 行注释；
- Wordwrap 自动换行。

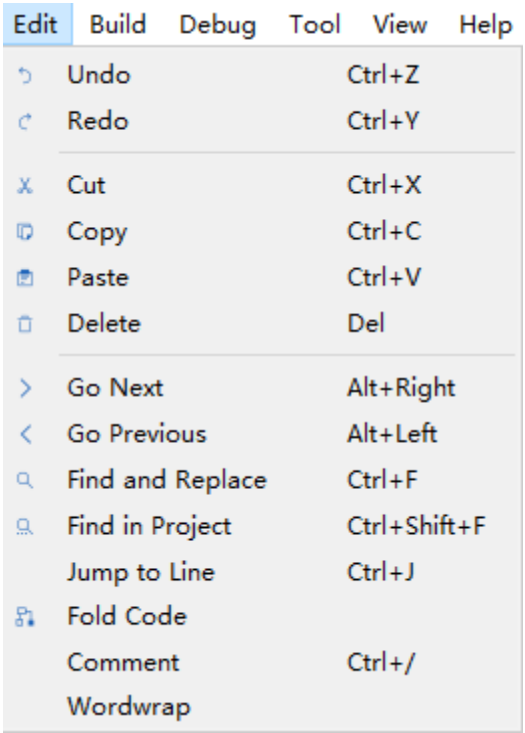
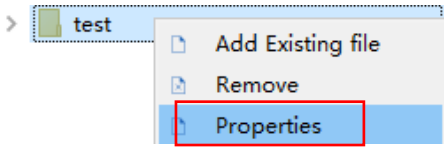


图 2-4 Edit 编辑菜单

5. 编译一个工程或者链接库之前/之后执行操作可通过选中 FileSystem 栏下项目名称，然后点击鼠标右键，选择 Properties 如图所示，在弹出的界面图中进行相关设置。



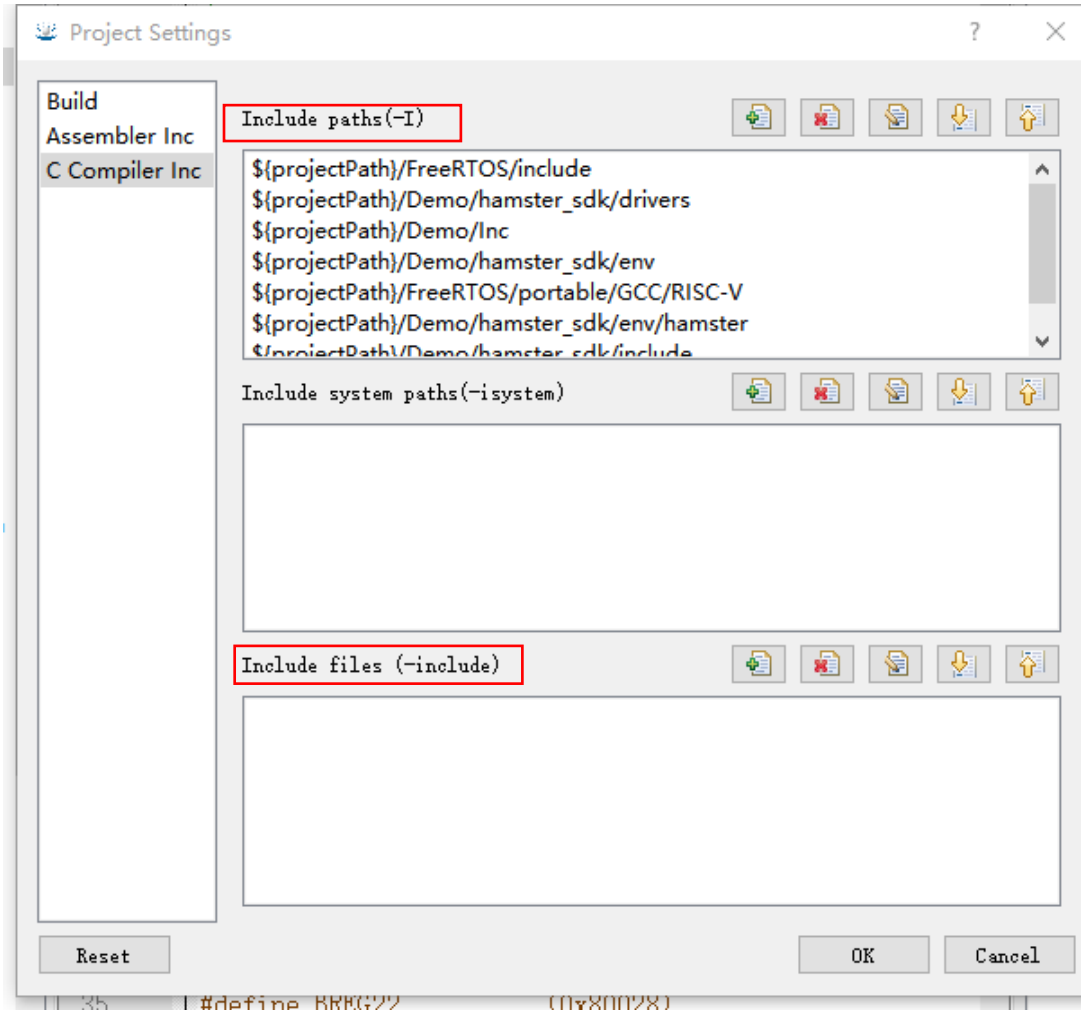


图 2-5 属性配置

6. 对项目进行编译操作，点击菜单栏 Build，如图 2-5 所示。
   - ✧ Build 对上次更改过的文件进行编译；
   - ✧ Clean 清理中间文件；
   - ✧ Rebuild 编译工程中所有源文件；
   - ✧ Download 编译成功后，可通过 download 下载到开发板运行，点击 download,弹出 Robei IDE 下载界面，如图 2-6 所示。

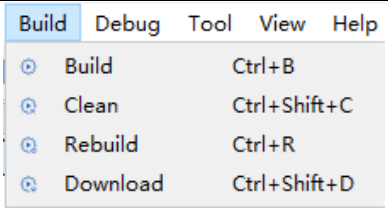


图 2-5 Build 菜单

图 2-6 下载界面

7. 对开发板中运行程序进行调试，在菜单栏中选择 Debug，如图 2-7 所示。

- Setup debugger 配置调试器；
- Start debugger 启动调试器；
- Stop 停止调试；
- Continue 继续调试，执行至下一个断点；
- Step over 单步跳过，从断点处开始，执行单步语句；
- Step into 单步调试，可查看当前执行位置的详细情况；
- Step out 单步跳出，与 Step into 相反；
- Force stop 强制停止调试。

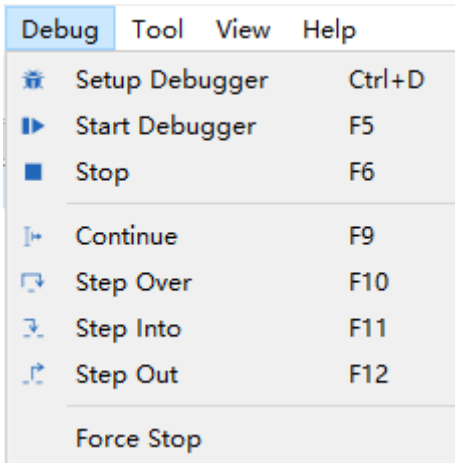


图 2-7 Debug 菜单

8. 设置首选项，偏好设置，点击菜单栏的 Tool—>Preference 进行设置，弹出如图所示界面。

- 在 editor 栏下可进行文本编辑器设置：
  - Save files on compile：在编译时保存文件，

- Show spaces in editor：在编辑器中显示空格，
- Replace tabs with space：用空格替换 tab，
- Detect file identation：打开文件时，自动检测文件内容，
- Display line number：显示行数。

- 在 paths 栏下设置偏好路径；
- 在 others 栏下进行其他设置。

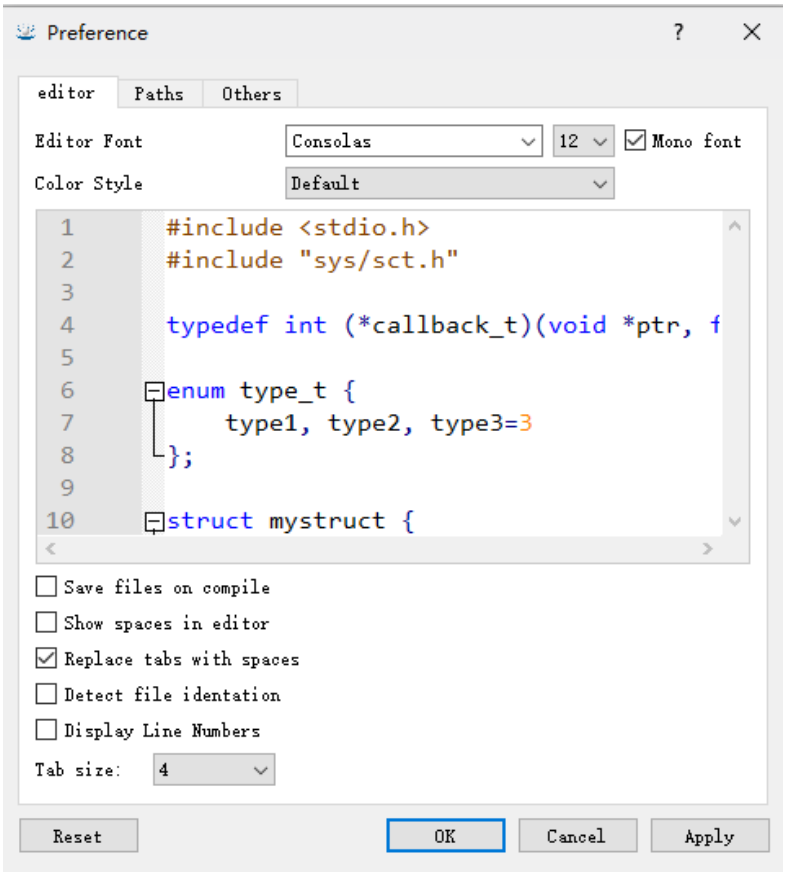


图 2-8 Preference 对话框

9. 点击菜单栏的 View 可进行自定义菜单栏设置，如图所示。

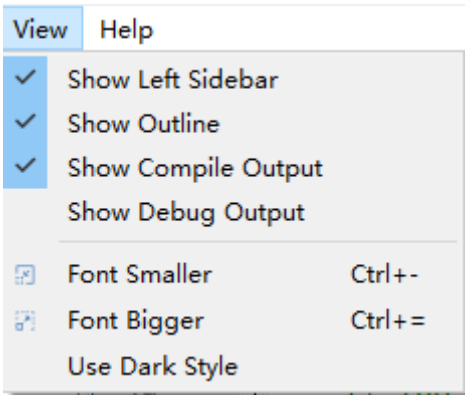


图 2-9 View 菜单

3. GPIO 介绍

3.1 Robei GPIO 简介

沉芯芯片支持通过软件对内部虚拟引脚按照组进行重构（Remap），每个组八个引脚，可以和物理引脚进行分组映射，比如代表虚拟引脚的 GPA，可以映射到物理的引脚 C7~C0，代表虚拟引脚的 GPD,可以映射到物理引脚 B7~B0。每组映射比特位也相互对应，如 GPA0 对应 C0， GPA5 对应 C5 等。

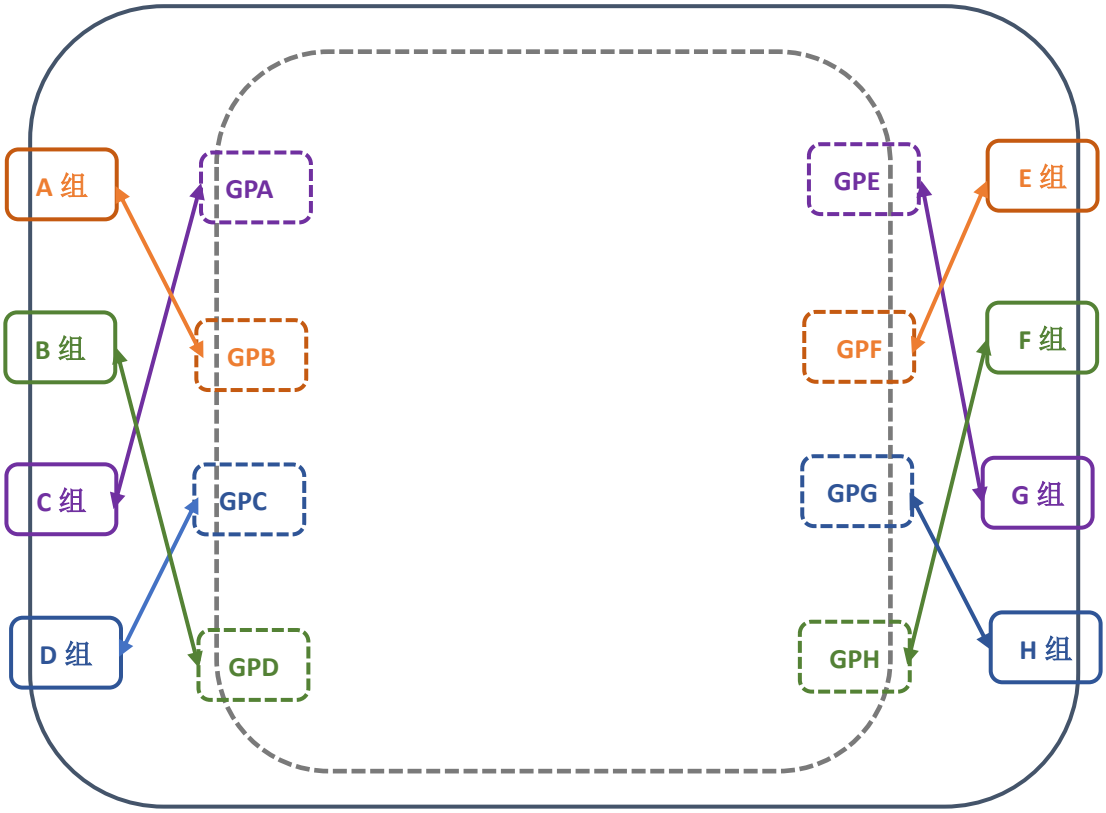


图 3-1 沉芯芯片内部虚拟引脚与物理引脚按组重构（Remap）

3.2 点亮 LED 灯实验

实验简述：

通过设置 GPIO 的高低来点亮板卡上的贴片 LED 灯。

操作流程：

1. 首先是初始化 GPIO 引脚，本案例采用 C 组引脚中的第 0 号引脚作为 LED 灯的个控制信号；
2. 通过设置高电平，点亮 LED 灯；

3. 通过设置低电平，熄灭灯；
4. 往复循环，实现灯的闪烁；

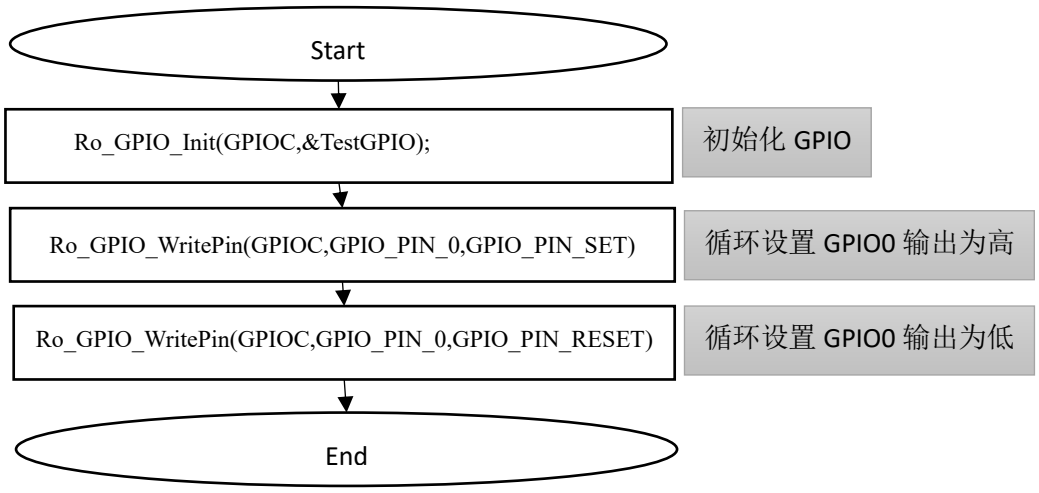


图 3.2 点亮 LED 灯函数流程

示例代码：

```
int main( void )
{
     GPIO_InitTypeDef TestGPIO;              //选中 C 组 Pin 0 来点亮 LED
    TestGPIO.Pin= GPIO_PIN_0;                //设置为输出模式
    TestGPIO.Mode=GPIO_MODE_OUTPUT;
    Ro_GPIO_Init(GPIOC,&TestGPIO);       //初始化 GPIO

    while(1)
     {
          if(i==400000)
          {
                //设置 GPIOC Pin0 输出为高，点亮 LED
              Ro_GPIO_WritePin(GPIOC, GPIO_PIN_0, GPIO_PIN_SET);
          }
           else if(i==800000)
           {
                 //设置 GPIOC Pin0 输出为低，关掉 LED
              Ro_GPIO_WritePin(GPIOC, GPIO_PIN_0, GPIO_PIN_RESET);
                i=0;
            }
           i++;
    }
}
```

函数解释：

GPIO 初始化：

void Ro_GPIO_Init (GPIO_TypeDef *GPIOx, GPIO_InitTypeDef * GPIO_Init)；

参数： GPIOx： GPIO 对应的组，如 GPIOA-GPIOM

GPIO_Init：是一个结构体，GPIO 对应的管脚。其中，Pin 对应的是 GPIO_PIN_0-GPIO_PIN_7， Mode 对应的有：GPIO_MODE_INPUT;
GPIO_MODE_OUTPUT;
GPIO_MODE_IT_RISING;
GPIO_MODE_IT_FALLING;
GPIO_MODE_IT_RISING_FALLING

3.3 GPIO 输入输出实验

实验简述：

设置 GPIO D 组 Pin0 为输入，GPIO C 组 Pin0 为输出，然后通过跳线来短接 D0 C0,使 C0 循环输出高低，读取 D0，通过串口将其信息输出至电脑端。

操作流程：

首先设置 C 组的第 0 比特作为输出，设置 D 组的第 0 比特作为输入，然后通过 Ro_GPIO_WritePin 将 C 组的第 0 比特设置为高电平，此时读入 D 组 GPIO 的状态值，显示到电脑端。然后设置 C 组的 GPIO 值为低电平，读出 D 组的 GPIO 值。

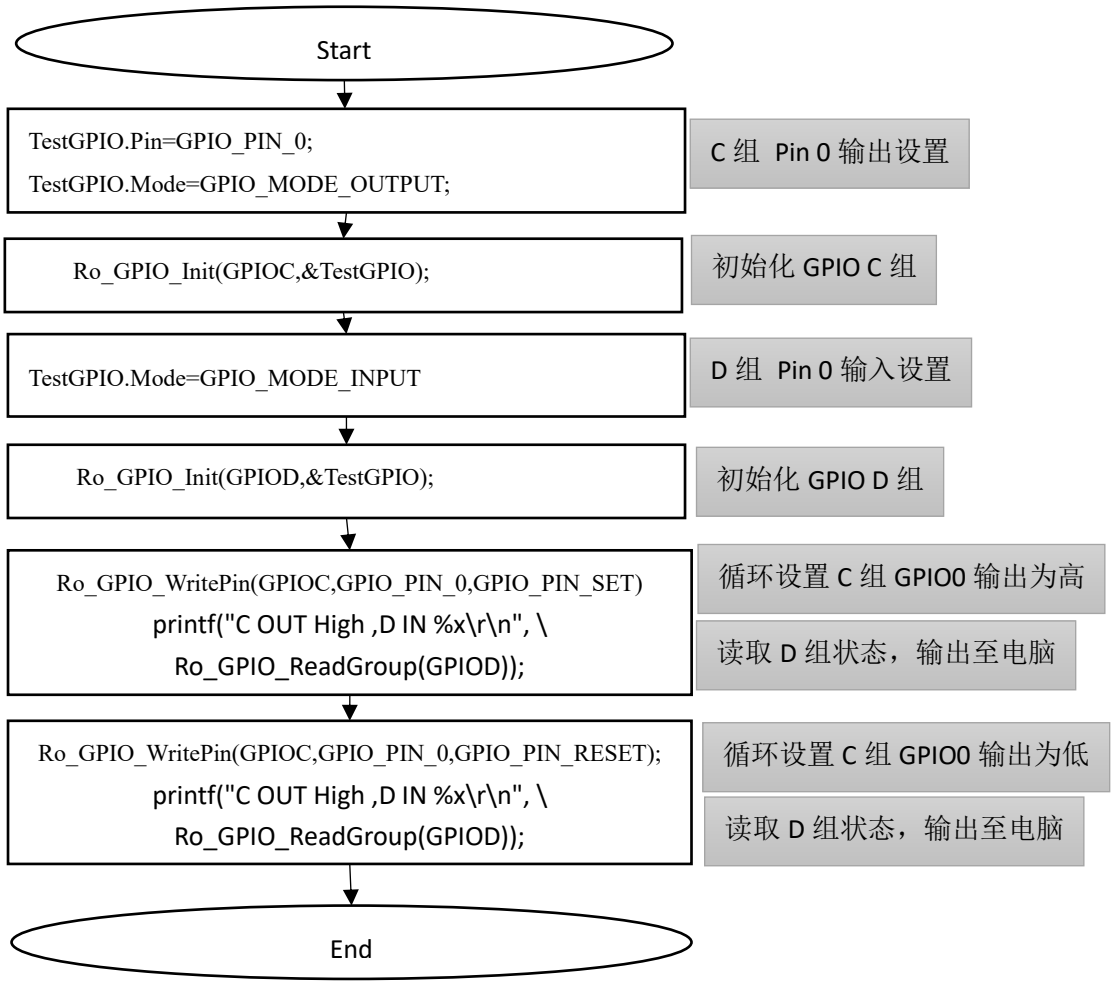


图 3.3 GPIO 输入输出流程图

示例代码：

```
void Ro_INOUT_testDinCout(void)
{
    GPIO_InitTypeDef TestGPIO;
    TestGPIO.Pin=GPIO_PIN_0;                    //设置 C 组的 Bit 0
    TestGPIO.Mode=GPIO_MODE_OUTPUT;     //设置 C 组的 GPIO 为输出
    Ro_GPIO_Init(GPIOC,&TestGPIO);              //初始化 IO

    TestGPIO.Mode=GPIO_MODE_INPUT;      //设置 D 组为输入
    Ro_GPIO_Init(GPIOD,&TestGPIO);
    Ro_GPIO_WriteGroup(GPIOC, 0xff);

    while(1)
    {
        if(i==100000)
        {
```

```
                Ro_GPIO_WritePin(GPIOC,GPIO_PIN_0,GPIO_PIN_SET);
                // 设置 C 组第 0 比特高电平
                   printf("C OUT High ,D IN %x\r\n", Ro_GPIO_ReadGroup(GPIOD));
            }
            else if(i==200000)
            {
                   i=0;
                Ro_GPIO_WritePin(GPIOC,GPIO_PIN_0,GPIO_PIN_RESET);
                // 设置 C 组第 0 比特低电平
                   printf("C OUT High ,D IN    %x\r\n", Ro_GPIO_ReadGroup(GPIOD));
            }
            i++;
      }
}
```

函数注释：

读取输入 GPIO 当前组的管脚的状态：

GPIO_PinState Ro_GPIO_ReadGroup(GPIO_TypeDef *GPIOx);

参数：GPIOx: GPIO 需要读取的组，GPIOA-GPIOM

GPIO_PinState 返回值为当前组 8 个 GIPO 管脚的状态.

3.4 GPIO 中断实验

实验简述：

设置 GPIO D 组 Pin0 为上升沿触发中断，Pin1 为下降沿触发，Pin2 为电平变化触发，GPIO C 组 Pin0 为循环输出高低电平，然后通过跳线来短接 D0 和 C0，通过串口将中断触发信息输出至电脑端。

操作流程：

先设置 C 组合 D 组的引脚配置，设置 D 组为下降沿触发，然后使能管脚中断，为 C 组第 0 比特设置输出高电平再设置为低电平。

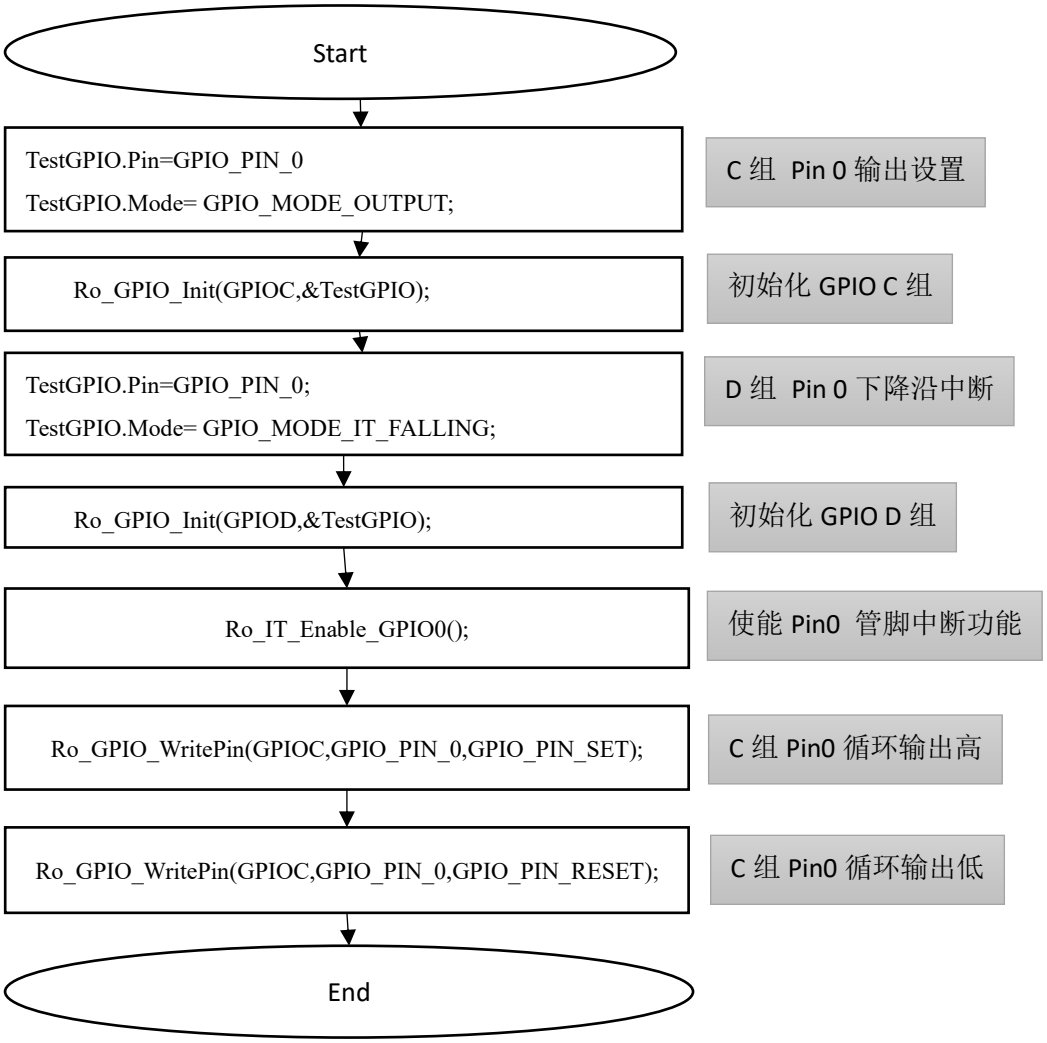


图 3.4 GPIO 中断实验流程图

示例代码：

```
void Ro_INOUT_testDinCout(void)
{
    GPIO_InitTypeDef TestGPIO;
    TestGPIO.Pin=GPIO_PIN_0;
    TestGPIO.Mode=GPIO_MODE_OUTPUT;
    Ro_GPIO_Init(GPIOC,&TestGPIO);
    //下降沿触发中断使用        GPIO_MODE_IT_FALLING;
    //上升沿触发中断使用        GPIO_MODE_IT_RISING
    //电平变化触发中断使用            GPIO_MODE_IT_RISING_FALLING
    TestGPIO.Mode= GPIO_MODE_IT_FALLING;
    Ro_GPIO_Init(GPIOD,&TestGPIO);
    Ro_GPIO_WriteGroup(GPIOC, 0xff);
    while(1)
```

```
    {
        if(i==100000)
        {
            Ro_GPIO_WritePin(GPIOC,GPIO_PIN_0,GPIO_PIN_SET);
        }
        else if(i==200000)
        {
            i=0;
            Ro_GPIO_WritePin(GPIOC,GPIO_PIN_0,GPIO_PIN_RESET);
        }
        i++;
    }
}
```

3.5 GPIO 映射实验

实验简述：

设置 GPIO GPC 组映射（Remap）到物理引脚 D 组，然后 GPC 组输出高低电平，用万用表或者示波器测量 D 组变化。

操作流程：

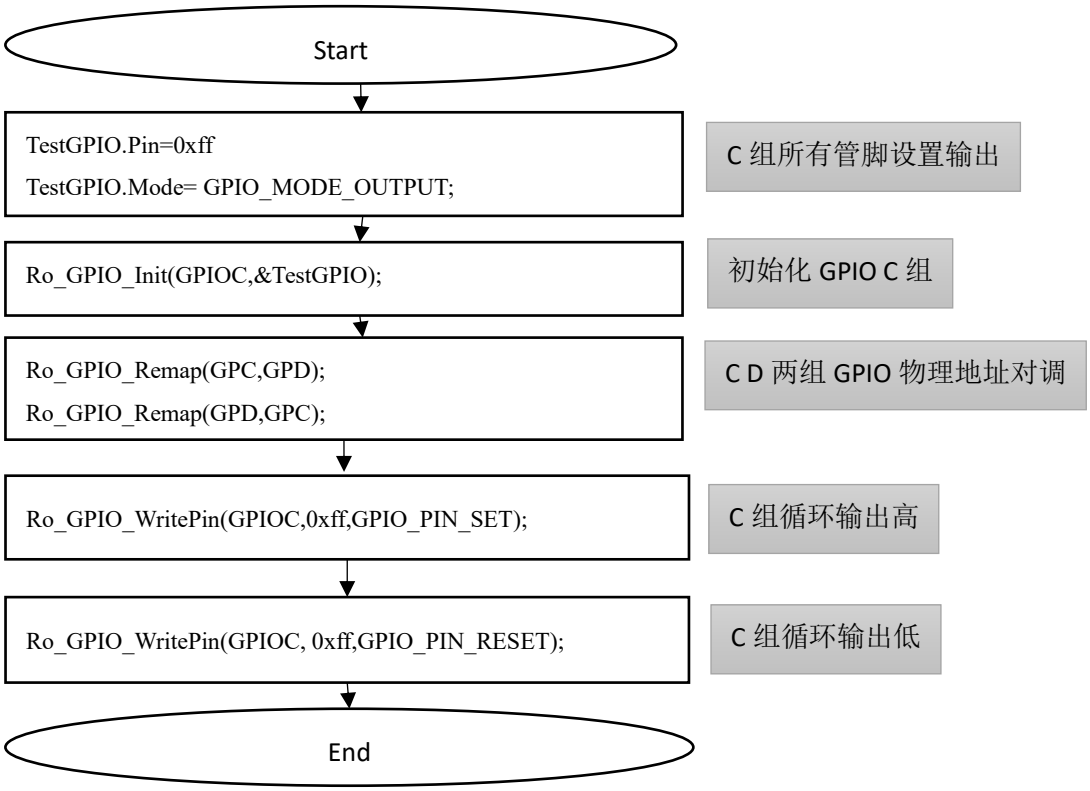


图 3.5 GPIO 映射实验流程图

示例代码：

```
void Ro_CD_Remap(void)
{
    GPIO_InitTypeDef TestGPIO;
    TestGPIO.Pin=GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3|
                 GPIO_PIN_4|GPIO_PIN_5| GPIO_PIN_6| GPIO_PIN_7;
    TestGPIO.Mode=GPIO_MODE_OUTPUT;
    Ro_GPIO_Init(GPIOC,&TestGPIO);

    Ro_GPIO_WriteGroup(GPIOC, 0xff);
    while(1)
    {
        if(i==100000)
        {
            Ro_GPIO_WritePin(GPIOC,0xFF,GPIO_PIN_SET);
        }
        else if(i==200000)
        {
                i=0;
            Ro_GPIO_WritePin(GPIOC,0xFF,GPIO_PIN_RESET);
        }
        i++;
    }
}
```

4.看门狗实验

4.1 看门狗简介

看门狗负责定期的查看芯片内部的情况，一旦发生错误就向芯片发出重启信号。看门狗在程序的中断中拥有最高的优先级，防止程序跑飞，也可以防止程序在线运行的时候出现死循环。而窗口看门狗（WWDG）通常被用来监测应用程序背离原有的运行序列导致的故障，而这些故障多由外部干扰或内部不可预见的逻辑条件所导致的。在使用上，需要用户在一个固定时间来喂狗，一旦超时将触发中断，如同用户养的狗，要固定时间喂食，否则狗就会饿死。

窗口看门狗就是一个倒数计数装置，在一定条件下可以重置。与这个倒数计数装置相关的两个基本变量是 WWDG_COUNT 和 WWDG_WR，WWDG_COUNT 代表了重置后倒计数的初始值（递减计数器的初始值），WWDG_WR 代表可以重置的最大数值（递减计数到这个值就可以喂狗了，也就是最早可以喂狗的数值），用户可根据需要在宏定义中更改它们的值从而改变计数范围。窗口看门狗的原理如下图所示。

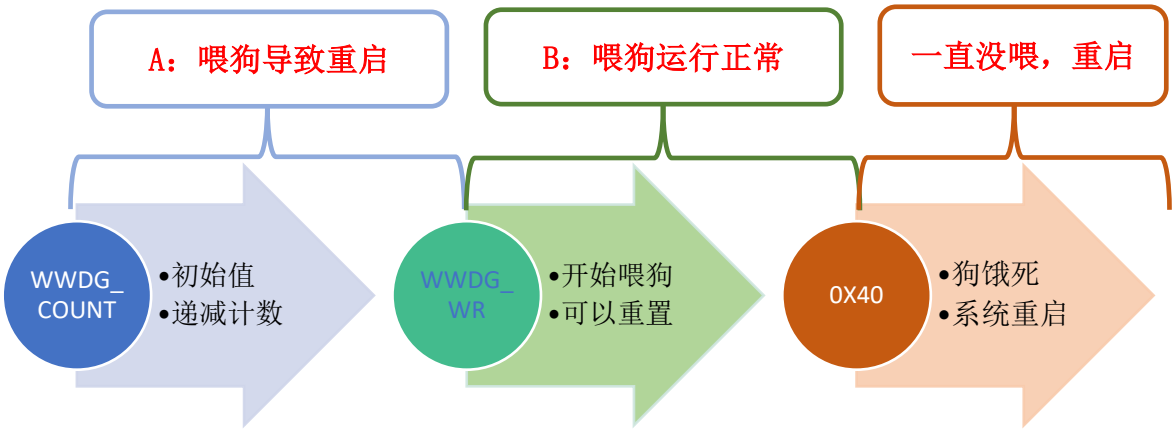


图 4-1 窗口看门狗计数原理

简单地讲：三个数值（WWDG_COUNT、WWDG_WR 和 0x40），在启动后，看门狗就开始从 WWDG_COUNT 进行递减计数，直到计数到 0x40（ CPU 规定的最低数值）。由于规定在计数到达 WWDG_WR 之前，不能 refresh 看门狗（亦即“喂狗”），在计数值小于 WWDG_WR 内才允许喂狗，所以计数器在大于 WWDG_WR 时喂狗就会导致系统重启；一直都没有喂狗（计数记到了 0x40，狗

狗饿死了）会导致系统重启；计数在小于 WWDG_WR 但是大于 0x40 范围时触发中断并在中断服务函数内喂狗系统会正常运行。

4.2 窗口看门狗实验

实验简介：

打开看门狗之后，持续喂狗，保障看门狗不会被饿死。

操作流程：

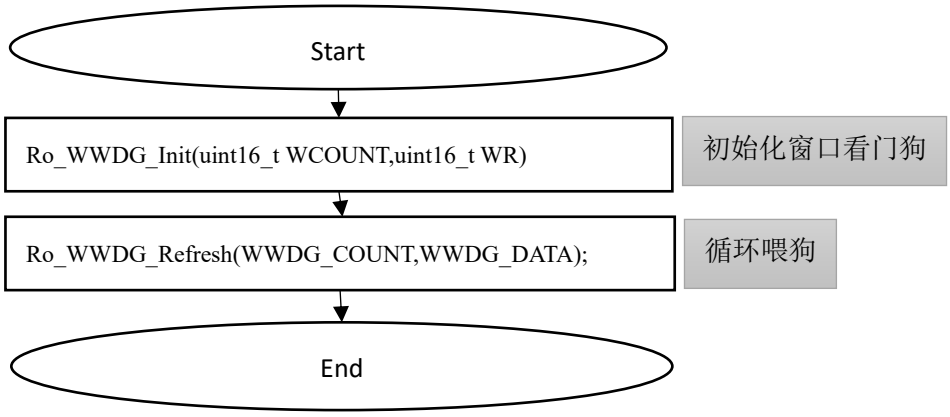


图 4-2 看门狗操作流程

示例代码：

```
int main( void )
{
    huart0.Instance=UART0;
    Ro_Init_Uart(&huart0);
    huart1.Instance=UART1;
    Ro_Init_Uart(&huart1);

    Ro_WWDG_Init(WWDG_COUNT,WWDG_DATA);
    while(1)
    {
        Ro_WWDG_Refresh(WWDG_COUNT,WWDG_DATA);
        uart0_putchar('F');
    }
}
```

函数注释：

void Ro_WWDG_Init(uint16_t WCOUNT,uint16_t WR)

- ✓ WCOUNT 窗口计数器，必须大于 0x40，最大值 0x3fff
- ✓ WR 喂狗范围，在 0x40-WR 之间可喂狗，WR<=WCOUNT

实验结果：

```
TEST Hamster_WWDG_nReset
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

图 4-3 窗口看门狗运行结果

5. TIMER 实验

5.1 Timer 简介

Timer（计数器）可以用于脉宽调制（PWM）、计时等，沉芯芯片有 1 个 32 位的计数器 TM32 和 4 个 16 位的计数器 TM16，并且每个计数器都有对应的 4 组 PWM 和 nPWM（PWM 波形的反向，PWM 为 1 时，nPWM 为 0）。

表 5-1 可用于 PWM 的虚拟 GPIO 映射

虚拟引脚组名	编号	支持重构功能
GPB	0	TM16_0_PWM0
	1	TM16_0_nPWM0
	2	TM16_0_PWM1
	3	TM16_0_nPWM1
	4	TM16_0_PWM2
	5	TM16_0_nPWM2
	6	TM16_0_PWM3
	7	TM16_0_nPWM3
GPD	0	TM32_PWM0
	1	TM32_nPWM0
	2	TM32_PWM1
	3	TM32_nPWM1
	4	TM32_PWM2
	5	TM32_nPWM2
	6	TM32_PWM3
	7	TM32_nPWM3
GPF	0	TM16_1_PWM0
	1	TM16_1_nPWM0
	2	TM16_1_PWM1
	3	TM32_PWM0
	4	TM32_PWM1
	5	TM32_nPWM0
	6	TM32_nPWM1
	7	TM16_1_nPWM1
GPG	0	TM16_1_PWM2
	1	TM16_1_nPWM2
	2	TM16_1_PWM3
	3	TM16_1_nPWM3
	4	TM16_2_PWM0
	5	TM16_2_nPWM0
	6	TM16_2_PWM1

	7	TM16_2_nPWM1
GPH	0	TM16_2_PWM2
	1	TM16_2_nPWM2
	2	TM16_2_PWM3
	3	TM16_2_nPWM3
	4	TM16_3_PWM0
	5	TM16_3_nPWM0
	6	TM16_3_PWM1
	7	TM16_3_nPWM1
GPI	0	TM16_3_PWM2
	1	TM16_3_nPWM2
	2	TM16_3_PWM3
	3	TM16_3_nPWM3
	4	TM16_0_Break
	5	TM16_1_Break
	6	TM16_2_Break
	7	TM16_3_Break

其中，TM32 对应 Timer0；TM16_0 对应 Timer16 的第一个；TM16_1 对应 Timer16 的第二个；TM16_2 对应 Timer16 的第三个；TM16_3 对应 Timer16 的第四个。

5.2 PWM 实验

函数定义：

void Ro_Timer16_pwm(TIM_HandleTypeDef *htim, uint8_t Remapflag)

函数流程：

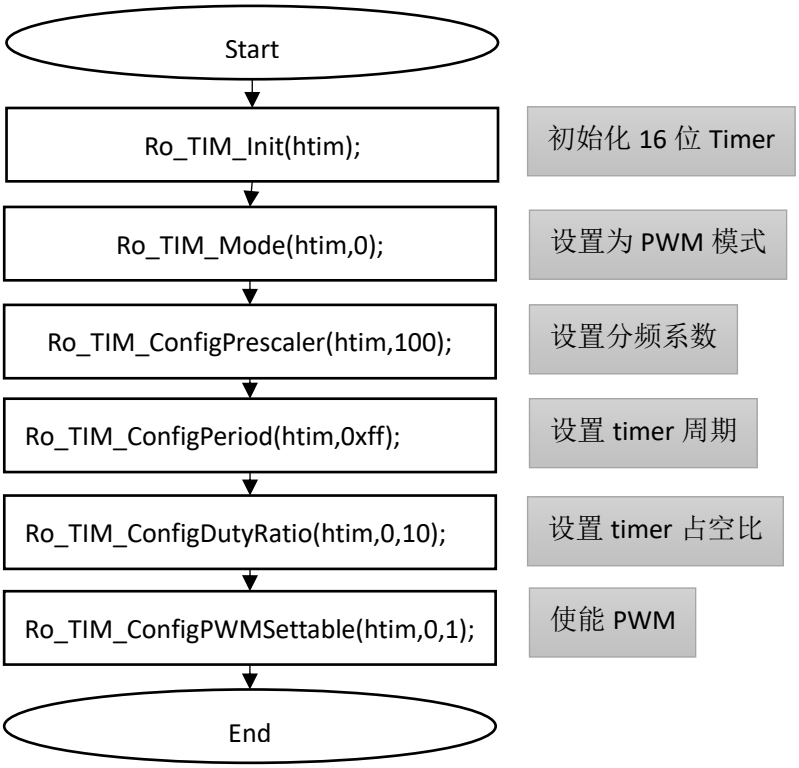


图 5-1 PWM 流程图

示例代码：

```
void Ro_Timer16_pwm(TIM_HandleTypeDef *htim,uint8_t Remapflag)
{
    Ro_TIM_Init(htim);
    //配置  timer16 的工作状态为计时器或 pwm
    Ro_TIM_Mode(htim, TIM_MODE_PWM);
   //配置分频系数
    Ro_TIM_ConfigPrescaler(htim,100);
    Ro_TIM_ConfigPeriod(htim,0xff);
    //设置 timer 四路 pwm 30%的占空比
    Ro_TIM_ConfigDutyRatio(htim,0,30);
    Ro_TIM_ConfigDutyRatio(htim,1,30);
    Ro_TIM_ConfigDutyRatio(htim,2,30);
    Ro_TIM_ConfigDutyRatio(htim,3,30);
    //使能 PWM
    Ro_TIM_ConfigPWMSettable(htim,0,1);
    Ro_TIM_ConfigPWMSettable(htim,1,1);
    Ro_TIM_ConfigPWMSettable(htim,2,1);
    Ro_TIM_ConfigPWMSettable(htim,3,1);
}
```

函数注释：

1. 设置 Timer16 的工作状态：

void Ro_TIM_Mode (TIM_HandleTypeDef *htim, uint8_t u8TMMode)

参数： htim　　Timer 的指针；

u8TMMode　有三种工作模式，分别为 TIM_MODE_PWM，TIM_MODE_TIMING 和 TIM_MODE_CAPTURE

2. 配置 Timer16 的分频系数

void Ro_TIM_ConfigPrescaler (TIM_HandleTypeDef *htim,uint16_t psc)

参数： htim　　Timer 的指针；

Psc　　时钟分频的系数

3. 使能或禁用 timer16 的某通道的 pwm

void Ro_TIM_ConfigPWMSettable (TIM_HandleTypeDef *htim,
uint8_t pwm_n, uint8_t status)

参数： htim　　Timer 的指针；

pwm_n　　要配置的 PWM 通道序号

status　　要把 pwm 通道配置为状态，0 禁用，非 0 使能

4. 配置 Timer16 的计数器初始值和结束值

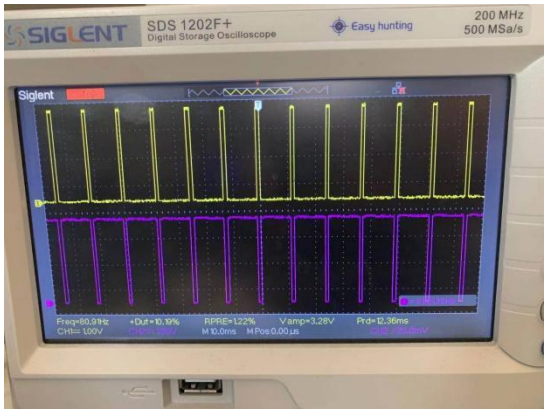
void Ro_TIM_ConfigPeriod (TIM_HandleTypeDef *htim, uint32_t cnt)

参数： htim　　Timer 的指针；

cnt　　总的计数个数

实验结果：

占空比：30 的情况



占空比：90 的情况

图 5-2 PWM 波形展示

5.2 DeadTime 实验

函数定义：

Ro_Timer16_deadtime(&htim1,0)为 PWM 的死区测试，先初始化配置 0-7 号管脚，定义死区时间；然后根据 Remapflag 判断 TM16 是否要 Remap，这里不需要；配置模式为测试 PWM，初始化预分频系数和周期。

函数流程:

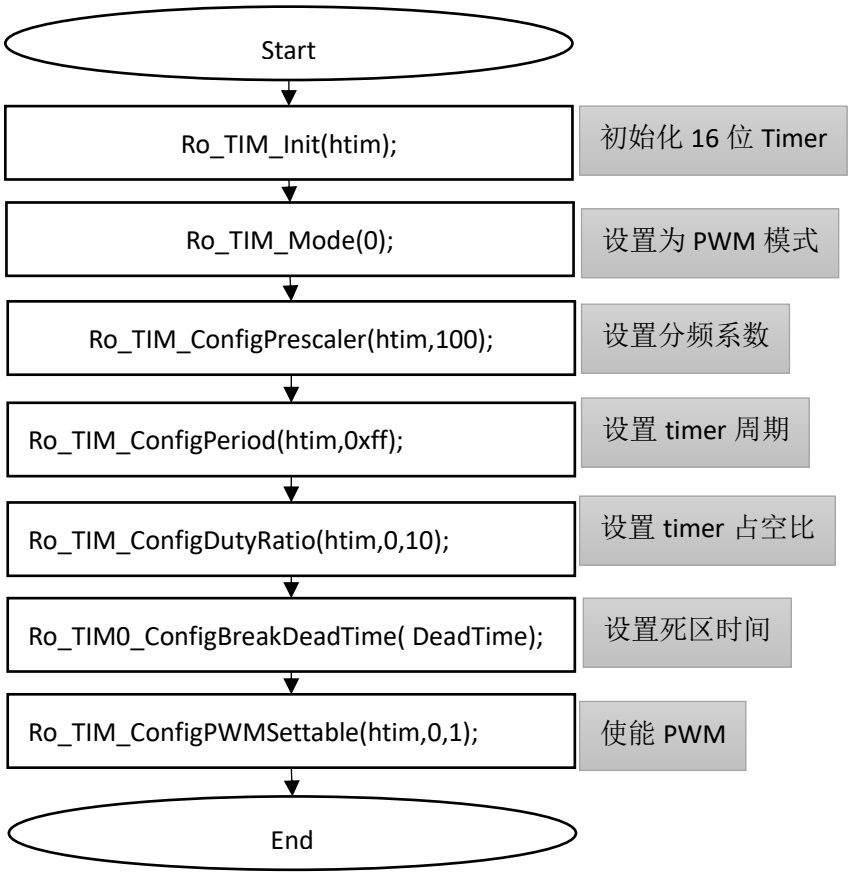


图 5-3 DeadTime 流程

示例代码：

```
void Ro_Timer16_deadtime(TIM_HandleTypeDef *htim,uint8_t Remapflag)
{
    //设置为 PWM 模式
    Ro_TIM_Mode(htim, TIM_MODE_PWM);
    Ro_TIM_ConfigPrescaler(htim,100);
    Ro_TIM_ConfigPeriod(htim,0xff);
    //设置 timer 四路 pwm 30%的占空比
    Ro_TIM_ConfigDutyRatio(htim,0,30);
    Ro_TIM_ConfigDutyRatio(htim,1,30);
```

```
    Ro_TIM_ConfigDutyRatio(htim,2,30);
    Ro_TIM_ConfigDutyRatio(htim,3,30);
    //设置 deadtime
    Ro_TIM_ConfigBreakDeadTime(htim,DeadTime);
    //使能 PWM
    Ro_TIM_ConfigPWMSettable(htim,0,1);
    Ro_TIM_ConfigPWMSettable(htim,1,1);
    Ro_TIM_ConfigPWMSettable(htim,2,1);
    Ro_TIM_ConfigPWMSettable(htim,3,1);
}
```

函数注释：

1. 配置死区时间：

void Ro_TIM_ConfigBreakDeadTime (TIM_HandleTypeDef *htim, uint8_t DeadTime)

参数： htim Timer 的指针；

DeadTime 时间范围 0x00-0xff

5.3 计时器实验

函数流程：

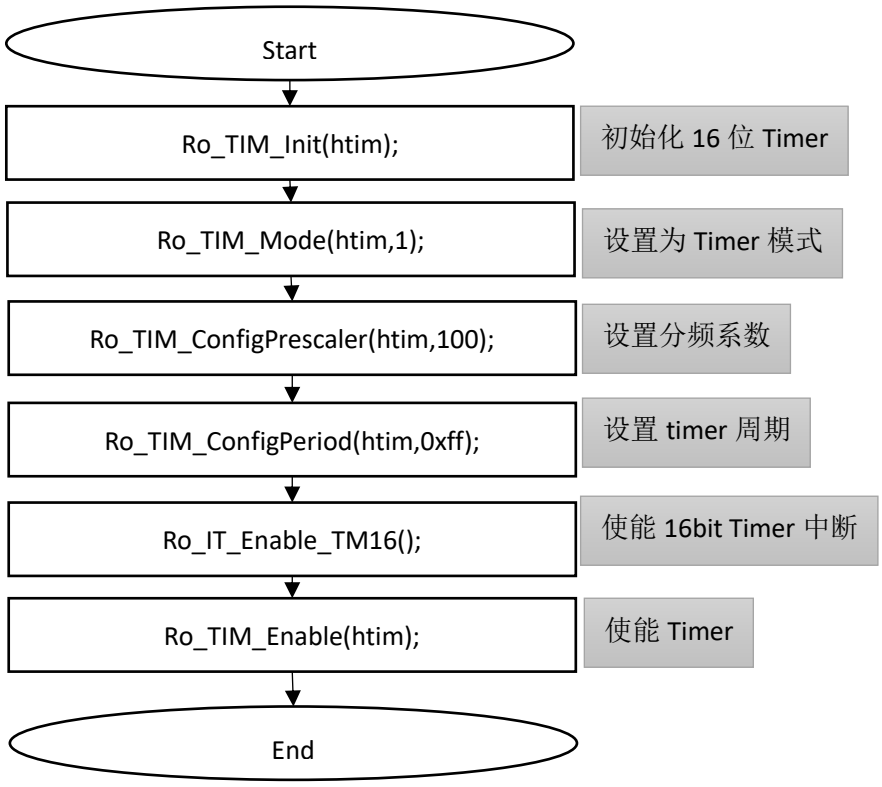


图 5-4 DeadTime 流程

示例代码：

```
void Ro_Timer16_tim(TIM_HandleTypeDef *htim)
{
    Ro_TIM_Init(htim);
    //设置为 Timer 模式
    Ro_TIM_Mode(htim, TIM_MODE_TIMING);

    Ro_TIM_ConfigPrescaler(htim,0xff);
    Ro_TIM_ConfigPeriod(htim,0xff);

    //使能 16bit Timer 中断
    Ro_IT_Enable_TM16();
    //使能 Timer 功能
    Ro_TIM_Enable(htim);
}
```

5.4 刹车功能实验

函数流程：

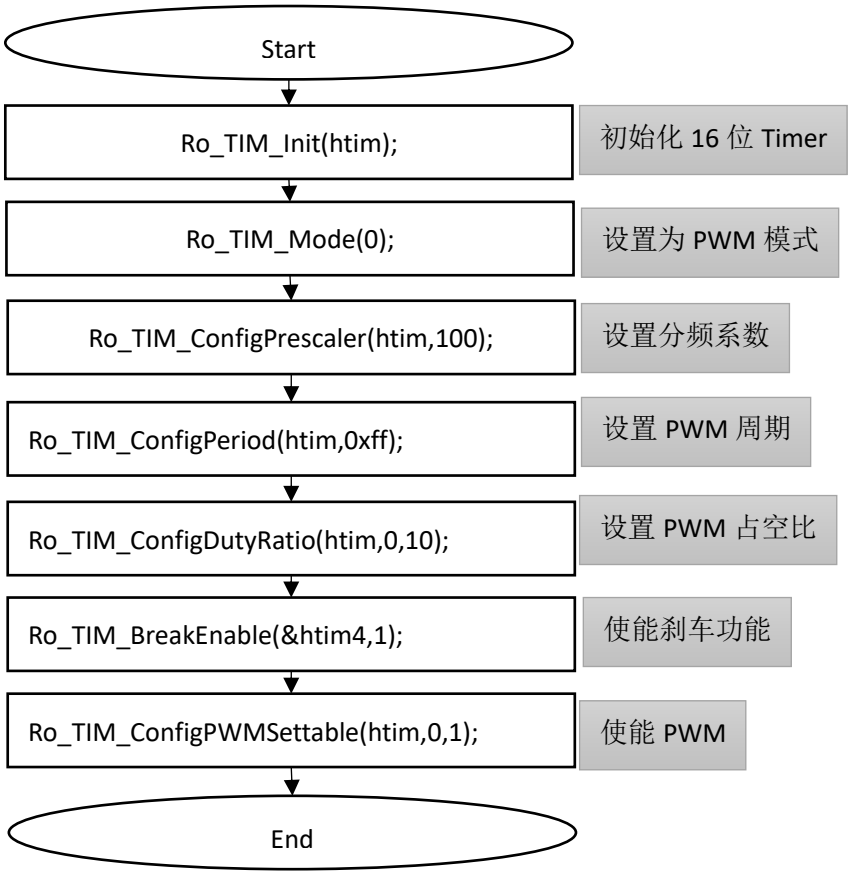


图 5-5 刹车流程

示例代码：

```
void Ro_Timer4_Break(void)
{
    Ro_TIM_Init(&htim4);
    Ro_TIM_Mode(&htim4,0);
    Ro_TIM_ConfigPrescaler(&htim4,0xff);
    Ro_TIM_ConfigPeriod(&htim4,100);

    Ro_TIM_ConfigDutyRatio(&htim4,0,30);   //check
    Ro_TIM_ConfigDutyRatio(&htim4,1,30);
    Ro_TIM_ConfigDutyRatio(&htim4,2,30);   //check
    Ro_TIM_ConfigDutyRatio(&htim4,3,30);

    Ro_TIM_BreakEnable(&htim4,1);    //使能有刹车功能
    Ro_TIM_ConfigPWMSettable(&htim4,0,1);
    Ro_TIM_ConfigPWMSettable(&htim4,1,1);
    Ro_TIM_ConfigPWMSettable(&htim4,2,1);
    Ro_TIM_ConfigPWMSettable(&htim4,3,1);
```

}

函数注释：

1. 使能 Timer 的刹车功能

void Ro_TIM_BreakEnable (TIM_HandleTypeDef *htim, uint8_t enable)

参数： htim Timer 的指针；

Enable 使能刹车功能

6. MPU 实验

6.1 MPU 模块简介

一些嵌入式系统使用多任务的操作和控制，因此这些系统必须要有一种有效机制来保证正在运行的任务不破坏其他任务操作。内存保护单元（MPU）就是有效保护系统资源的一种硬件，主要提供了内存区域的保护作用。

在本案例中，MPU 模块操作需要使用函数

void Ro_MPU_Setup (uint32_t Addr, uint32_t Area, uint8_t Size, uint8_t Type)

其中，

- ✧ Addr 为想要保护的地址；
- ✧ Area 为存放保护数据的地址，分为 MPU_AREA1 到 7，
- ✧ Size 为需要保护的内存大小，范围从 MPU_4K 到 MPU_2G；
- ✧ Type 为允许的操作，其中 MPU_UNRW 为不可读写、MPU_ONLYR 为只可读不可写、MPU_ONLYW 为只可写不可读以及 MPU_RW 为可读可写。

调用实例：

Ro_MPU_Setup(GPIO_BASE,MPU_AREA6,MPU_4K,MPU_UNRW);

对 GPIO 进行保护，范围是从 GPIO 的起始地址开始，利用 MPU 模块的 AREA6，保护 4K 长度，并且把操作限制为不可读写。如果对这段存储空间进行读或者写操作，就会引起 Crash，触发中断。

6.2 MPU 模块软件设计

函数流程：

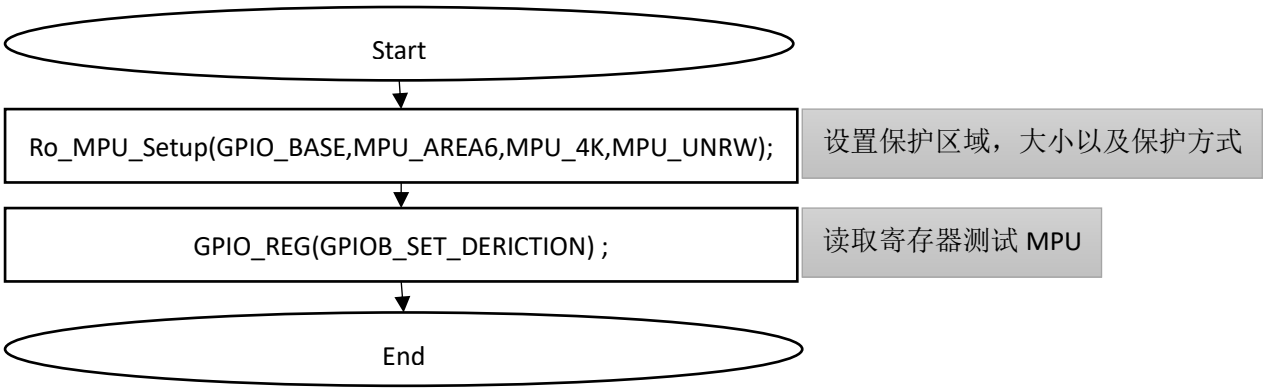


图 6-1 MPU 流程图

示例代码：

void Ro_MPU_UNRW(uint8_t u8Read)

```
{
    Ro_MPU_Setup(GPIO_BASE,MPU_AREA6,MPU_4K,MPU_UNRW);
    if(u8Read)
    {
        GPIO_REG(GPIOB_SET_DERICTION);
    }
    else
    {
         GPIO_REG(GPIOB_SET_DERICTION) =  0xdf;
    }
}
```

函数注释：

1. MPU 设置：

void Ro_MPU_Setup(uint32_t Addr,uint32_t Area,uint8_t size,uint8_t type)

参数设置：

- ✓ Addr：要保护的地址
- ✓ Area：存放保存数据的地址，可以设置的参数为 MPU_AREA1 到 MPU_AREA7
- ✓ size：需要保护的存储大小，例 MPU_4K，MPU_8K，MPU_16K……
- ✓ type: 允许的操作, 有以下几种：
  - MPU_UNRW不允许读写
  - MPU_ONLYR只允许读
  - MPU_ONLYW　只允许写
  - MPU_RW　　允许读写

7. DMA 实验

7.1 DMA 简介

DMA(Direct Memory Access，直接存储器访问) 是所有现代电脑的重要特色，它允许不同速度的硬件装置进行信息传递，而不需要依赖于 CPU 进行搬运。虽然 CPU 可以从一个存储器把一个一个的数据复制到暂存器，然后把它们写到新的地方，但是数据量比较大的时候，CPU 被完全占用，对其他的请求就无法响应。DMA 可以将数据从一个地址空间复制到另外一个地址空间，无须占用 CPU 的运行资源。CPU 只需要初始化这个传输动作，传输动作本身是由 DMA 控制器来实行和完成。数据传输操作并没有让处理器工作拖延，反而可以被重新排程去处理其他的任务。DMA 传输对于高效能嵌入式系统算法和网络是很重要的。

7.2 DMA 软件设计

函数流程：

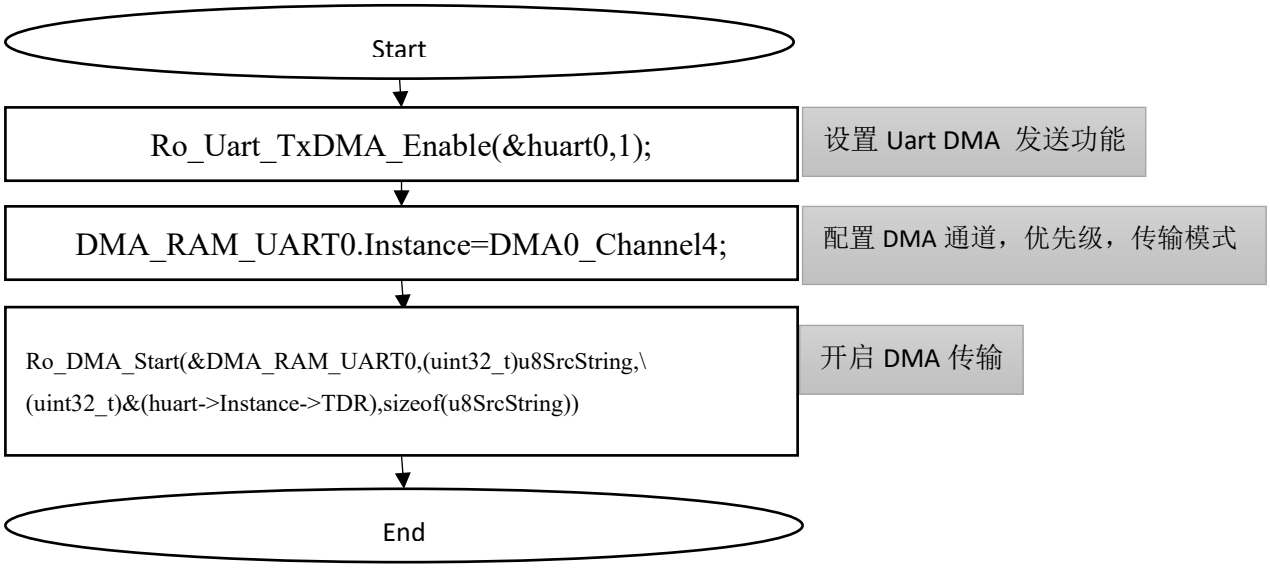


图 7-1 DMA 设计流程

示例代码：

```
void Ro_DMA_RAM_UART0_Test (UART_HandleTypeDef    *huart)
{
      g_u8printsel=0;
      Ro_Uart_RxDMA_Enable(&huart0,0);
      Ro_Uart_TxDMA_Enable(&huart0,1);
      DMA_RAM_UART0.Instance=DMA0_Channel4;
      DMA_RAM_UART0.Init.Priority=DMA_PRIORITY_VERY_HIGH;
      DMA_RAM_UART0.Init.SrcType=DMA_DEFAULT;
      DMA_RAM_UART0.Init.DstType=DMA_UART;
      DMA_RAM_UART0.Init.Mode=DMA_MODE_BYTE;
      Ro_Init_DMA(&DMA_RAM_UART0);
      Ro_DMA_Start(&DMA_RAM_UART0,(uint32_t)u8SrcString,
                  (uint32_t)&(huart->Instance->TDR), sizeof(u8SrcString));
      Ro_Uart_RxDMA_Enable(&huart0,0);
      Ro_Uart_TxDMA_Enable(&huart0,0);
}
```

函数注释：

1. 初始化 DMA：

void Ro_Init_DMA(DMA_HandleTypeDef *hdma);

通过 Power Control Register 打开 DMA Clock enable，打开 DMA overal config Register 的 CReq，使能 DMA;

2. 启动 DMA

void Ro_DMA_Start (DMA_HandleTypeDef *hdma,
uint32_t SrcAddress,
uint32_t DstAddress,
uint32_t DataLength);

参数说明：

- ✓ hdma dma 通道的 handle
- ✓ SrcAddress 源地址
- ✓ DstAddress 目的地址
- ✓ DataLength 数据长度

8. UART 案例

8.1 AS60X 指纹模块

8.1.1 AS603 简介

AS603 是一款高性能通用处理器芯片，采用“Cordis+5%2B”32 位 RISC 处理器内核，该内核具有 5 级深度流水线及专用 DSP 指令集，主频高达 128MHz，同时带有指纹功能。具有唯一的序列号，采用 Synochip 专利的安全设计方法，确保片内代码和数据安全。AS603 内置 128K 字节 SRAM、64K 字节 ROM 和 4K 位 OTPROM，外接 16M 字节 SQI FLASH、64M 字节异步 SRAM FLASH 和 16M 字节 SDRAM，满足大容量的数据和代码存储需求。具有丰富的外部接口，可满足复杂应用的场合。

指令/数据包共分为三类：

包标识=01 命令包

包标识=02 数据包，且有后续包

包标识=08 最后一个数据包，即结束包

所有的数据包都要加包头：0xEF01

01 命令包格式：

字节数	2bytes	4bytes	1 byte	2 bytes	1byte	… …			2 bytes
名称	包头	芯片地址	包标识	包长度	指令	参数 1	…	参数 n	校验和
内容	0xEF01	xxxx	01	N=					

02 数据包格式：

字节数	2bytes	4bytes	1 byte	2 bytes	N bytes…	2 bytes
名称	包头	芯片地址	包标识	包长度	数据	校验和
内容	0xEF01	xxxx	02			

08 结束包格式：

字节数	2bytes	4bytes	1 byte	2 bytes	N bytes…	2 bytes
名称	包头	芯片地址	包标识	包长度	数据	校验和
内容	0xEF01	xxxx	08			

- 数据包不能单独进入执行流程，必须跟在指令包或应答包后面。
- 下传或上传的数据包格式相同。
- 包长度 = 包长度至校验和（指令、参数或数据）的总字节数，包含校验和，但不包含包长度本身的字节数。
- 校验和是从包标识至校验和之间所有字节之和，超出 2 字节的进位忽略。
- 芯片地址在没有生成之前为缺省的 0xFFFFFFFF，一旦上位机通过指令生成了芯片地址，则所有的数据包都必须按照生成的地址收发。芯片将拒绝地址错误的数据包。

图 8-1 Synochip 公司官方的指令发送格式

8.1.2 AS603 软件设计

本案例通过 CPU 的 UART 模块的数据收发实现与 AS60X 指纹设备的交互通信，结合芯片的数据手册给出的指令详解（如下图）

(1) 录入图像　　PS_GetImage

- 功能说明： 探测手指，探测到后录入指纹图像存于 ImageBuffer。返回确认码表示：录入成功、无手指等。
- 输入参数： none
- 返回参数： 确认字
- 指令代码： 01H
- 指令包格式：

2 bytes	4bytes	1 byte	2 bytes	1 byte	2 bytes
包头	芯片地址	包标识	包长度	指令码	校验和
0xEF01	xxxx	01H	03H	01H	05H

- 应答包格式：

2 bytes	4bytes	1 byte	2 bytes	1 byte	2 bytes
包头	芯片地址	包标识	包长度	确认码	校验和
0xEF01	xxxx	07H	03H	xxH	sum

注：确认码=00H 表示录入成功；
确认码=01H 表示收包有错；
确认码=02H 表示传感器上无手指；
确认码=03H 表示录入不成功；
sum 指校验和

图 8-2 数据指令格式

✧ 要告诉设备录入指纹图像，需要发送的指令数据串为：
0xEF 0x01 0xff 0xff 0xff 0xff 0x01 0x00 0x03 0x01 0x00 0x05

✧ 如果录入成功，设备返回的数据串应该为：
0xEF 0x01 0xff 0xff 0xff 0xff 0x07 0x00 0x03 0x00　　sum

✧ 如果传感器上无手指，设备返回的数据串应该为：
0xEF 0x01 0xff 0xff 0xff 0xff 0x07 0x00 0x03 0x02　　sum

有了发送包格式和应答包格式之后，用户就可以按照指令包发送格式去编写测试函数实现包的发送了。

```
int main( void )
{
    huart0.Instance=UART0;
    Ro_Init_Uart(&huart0);
    huart1.Instance=UART1;
    Ro_Init_Uart(&huart1);
    Ro_AS60X()
    while (1){}
```

```
}
```

先初始化 UART 及其时钟，因为 AS60X 数据传输要求的 UART 波特率是 57600。我们别的模块波特率一般为 115200，所以对 UART 时钟的配置不一样，初始化 Ro_Init_Uart(); 里面设置 uart0/uart1 对应时钟和波特率。

```
void Ro_AS60X1to0( UART_HandleTypeDef *huart )
{
    g_u8printsel=0;
    g_u8UART1flag=0;
    uint32_t i=0;
    Ro_Uart_Enable(&huart1,1,1);
    Ro_Uart_RxDMA_Enable(&huart1,1);
    Ro_Uart_TxDMA_Enable(&huart1,0);
    Ro_IT_Enable_DMA01_CH4();
    DMA_UART1_RAM.Instance=DMA1_Channel4;
    DMA_UART1_RAM.Init.Priority=DMA_PRIORITY_VERY_HIGH;
    DMA_UART1_RAM.Init.SrcType=DMA_UART;
    DMA_UART1_RAM.Init.DstType=DMA_DEFAULT;
    DMA_UART1_RAM.Init.Mode=DMA_MODE_BYTE;
    Ro_Init_DMA(&DMA_UART1_RAM);
    DMA1_REG(DMA_OVERAL_CH_SELECT)=OVERAL_CHSEL_CS4(3);//使能 CR0
    printf("Start to send the order...\r\n");
    Ro_Uart_Reset(&huart1);

    uart0_putchar(0xef);
    uart0_putchar(0x01);
    uart0_putchar(0xff);
    uart0_putchar(0xff);
    uart0_putchar(0xff);
    uart0_putchar(0xff);
    uart0_putchar(0x01);
    uart0_putchar(0x00);
    uart0_putchar(0x03);
    uart0_putchar(0x01);
    uart0_putchar(0x00);
    uart0_putchar(0x05);

    Ro_DMA_AS60X(&DMA_UART1_RAM,(uint32_t)&(huart->Instance->RDR),(uint32_t)u
8AS60XDst,600);
    while(1)
    {
        if(i++==0xffff)
        {
```

```
                uart0_putchar(0xef);
                uart0_putchar(0x01);
                uart0_putchar(0xff);
                uart0_putchar(0xff);
                uart0_putchar(0xff);
                uart0_putchar(0xff);
                uart0_putchar(0x01);
                uart0_putchar(0x00);
                uart0_putchar(0x03);
                uart0_putchar(0x01);
                uart0_putchar(0x00);
                uart0_putchar(0x05);
                i=0;
            }
            if(g_u8UART1flag==1)
            {
                printf("Please _%d_: \r\n",__LINE__);
                uint16_t len=(READ_BIT(DMA1_REG(DMA_OVERALCONFIG),0xffff))+1;
               u8AS60XDst[599-len]=0;
                printf("Please _%d_: \r\n",__LINE__);

printf("\r\n%02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x\r\n",u8AS
60XDst[0],u8AS60XDst[1],u8AS60XDst[2],u8AS60XDst[3],u8AS60XDst[4],u8AS60XDst[5],u
8AS60XDst[6],u8AS60XDst[7],u8AS60XDst[8],u8AS60XDst[9],u8AS60XDst[10],u8AS60XDst
[11]);
                if(u8AS60XDst[9]==0x00)
                      printf("Registering fingerprint successfully!\r\n");
                else if(u8AS60XDst[9]==0x02)
                       printf("No fingerprint found!\r\n");
                else
                       printf("Unknown feedback...\r\n");
                g_u8UART1flag=0;
                Ro_Uart_Reset(&huart1);
                Ro_Uart_RxDMA_Enable(&huart1,1);
                Ro_Uart_TxDMA_Enable(&huart1,0);
                DMA1_REG(DMA_OVERAL_CH_SELECT)=OVERAL_CHSEL_CS4(3);
                Ro_DMA_AS60X(&DMA_UART1_RAM,(uint32_t)&(huart->Instance->RDR),\
                              (uint32_t)u8AS60XDst,600);
            }
        }
}
```

void Ro_AS60X1to0(UART_HandleTypeDef *huart)函数中在 DMA 搬运的基

础上实现了对 AS60X 指纹模块的驱动，在讲解代码实现之前，我们先要了解设备之间的硬件连接，连接顺序为：

UART0.txd->AS60X.rxd;AS60X.txd->UART1.rxd（3.3V 供电以及主从设备共地）。

首先将发送包用 uart0_putchar()的方式逐个发送出去，然后让 DMA 搬运处于中断模式等待数据接收，将接收到的数据搬运到一个数组之中，为了观察接收到的数据内容将它们打印出来。

8.1.2 AS603 实验结果

在 main.c 的主函数中添加 Ro_AS60X()函数，运行烧录后观察结果，如果不按手指上去，结果会显示“No fingerprint found!”字样；按上手指后会显示“Registering fingerprint successfully!”字样。

8.2 中科微 GPS 实验

8.2.1 GPS 模块简介

本案例基于中科微电子的 GPS 模块进行测试，与 UART 模块实验案例及其相关，建议使用者在实验完 UART 模块后再来运行本案例。与此同时，本案例也仅仅侧重于测试 CPU 对 GPS 模块的驱动能力，所以仅仅呈现了数据接收情况，GPS 模块其他的功能尚需使用者自行研究并且尝试去实现。

8.2.2 GPS 软件设计

一、硬件连线

设备遵从 UART 通信协议，因此对我们的主机来说只需要将一路 UART 与从设备的 VCC GND TXD（接 RXD） RXD（接 TXD）对应着连接好即可（PPS 管脚空出）。

二、测试用例讲解

与 UART 模块类似，通过 DMA 搬运的方式将从 UART1 接收到的 GPS 模块发来的信息搬到数组内，为了呈现接收结果，将数组的内容通过 UART0 打印出来。

main 函数里面调用 void Ro_GPSU1to0 函数，

```
int main( void )
{
    huart0.Instance=UART0;
    Ro_Init_Uart(&huart0);
    huart1.Instance=UART1;
    Ro_Init_Uart(&huart1);
```

```
    Ro_GPS();
}
```

先初始化 UART 及其时钟，因为中科微电子 GPS 数据传输要求的 UART 允许波特率有 9600，而别的模块的测试波特率一般为 115200，所以对 UART 时钟的配置不一样，初始化 Ro_Init_Uart();

```
void Ro_GPSU1toU0(UART_HandleTypeDef *huart)//20200810 ok
{
    g_u8printsel=0;
    printf("TEST %s \r\n",__FUNCTION__);
    //printf("Please enter the data by UART1: \r\n");
    Ro_Uart_Enable(&huart1,1,1);
    Ro_Uart_RxDMA_Enable(&huart1,1);
    Ro_Uart_TxDMA_Enable(&huart1,0);
    Ro_IT_Enable_DMA01_CH4();
    DMA_UART1_RAM.Instance=DMA1_Channel4;
     DMA_UART1_RAM.Init.Priority=DMA_PRIORITY_VERY_HIGH;
     DMA_UART1_RAM.Init.SrcType=DMA_UART;
     DMA_UART1_RAM.Init.DstType=DMA_DEFAULT;
     DMA_UART1_RAM.Init.Mode=DMA_MODE_BYTE;
     Ro_Init_DMA(&DMA_UART1_RAM);
    DMA1_REG(DMA_OVERAL_CH_SELECT)=OVERAL_CHSEL_CS4(3);//使能 CR0
     Ro_DMA_GPS(&DMA_UART1_RAM,&(huart->Instance->RDR),u8DestString,600);
    while(1)
    {
        if(g_u8UART1flag==1)
        {
            u8DestString[599-(READ_BIT(DMA1_REG(DMA_OVERALCONFIG),0xffff))]=0;
            printf("%s\r\n",u8DestString);
            g_u8UART1flag=0;
            Ro_Uart_Reset(&huart1);
            Ro_Uart_RxDMA_Enable(&huart1,1);
            Ro_Uart_TxDMA_Enable(&huart1,0);
            DMA1_REG(DMA_OVERAL_CH_SELECT)=OVERAL_CHSEL_CS4(3);
            Ro_DMA_TimeoutIT(&DMA_UART1_RAM,&(huart->Instance->RDR),u8DestString,600);
        }
    }
}
```

8.2.3 GPS 模块运行结果

在主函数中调用 Ro_GPS()函数运行后，可以在串口调试助手上看到不断打印出来的 GPS 地址信息，如图所示：

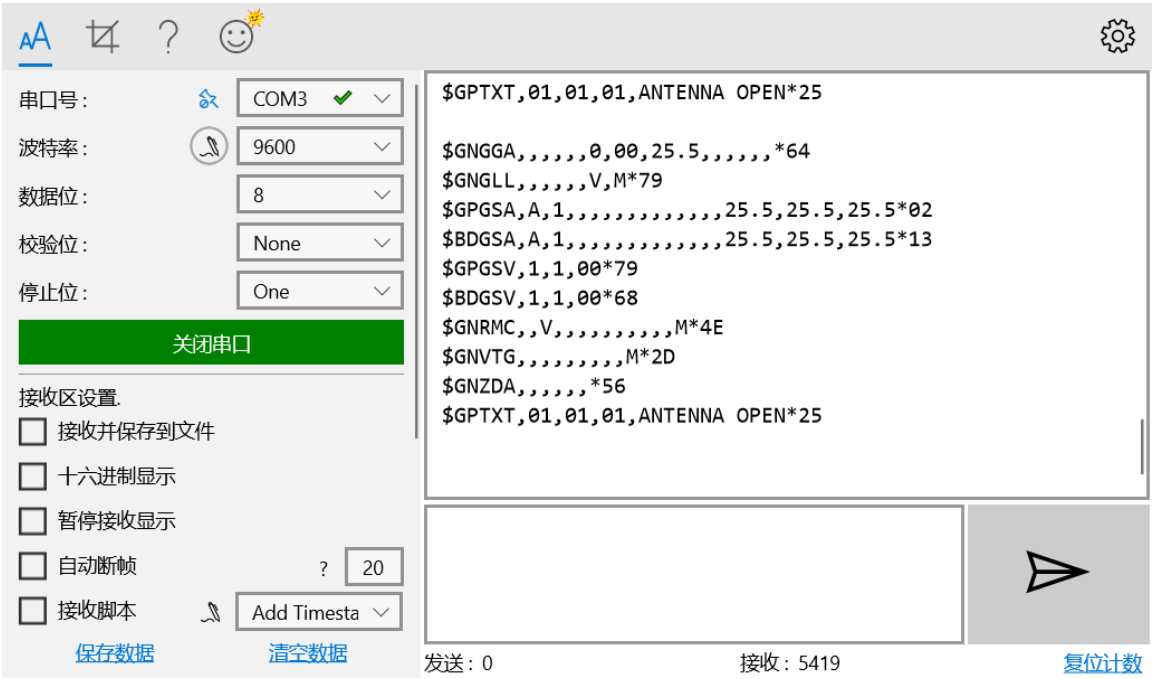


图 8-3 GPS 模块输出

注意：该设备支持的波特率为 9600，运行前要根据 CPU 时钟手册给出的频率分配方式计算一个 9600 波特率的配置方式进行初始化。

为了方便大家理解 GPS 发送的信息，下面附上简单的接受协议规范的讲解：

该设备服从 CASIC 多模卫星导航接收机协议规范，CASIC 接收机兼容国际标准 NMEA0183 协议，默认支持 NMEA0183 4.0 版本，兼容 V2.3 及 V3.X 版本，通过发送命令支持 NMEA0183 V4.1 标准，以及 V2.3 以前的标准。 数据以串行异步方式传送。第 1 位为起始位，其后是数据位。数据位遵循最低有效位优先的规则。

NMEA 消息由 GNSS 接收机发送，支持 NMEA0183 协议。数据格式协议框架

NMEA协议框架

检验和的计算范围

$	<地址>	{,<数值>}	*<校验和>	<CR><LF>

起始符	地址段	数据段	校验和段	结束序列
每条语句都是以‘$’开始	分为两部分：发送器标识符和语句类型	以‘，’开始，后面的数值长度是可变的，也有是定长的	对‘$’和‘*’之间的数据（不包括这两个字符）按字节进行异或运算的结果，用十六进制数值表示	每条语句都是以<CR><CF>结束

图 8-4 详细的 NMEA 协议标准参考 http://www.nmea.org/

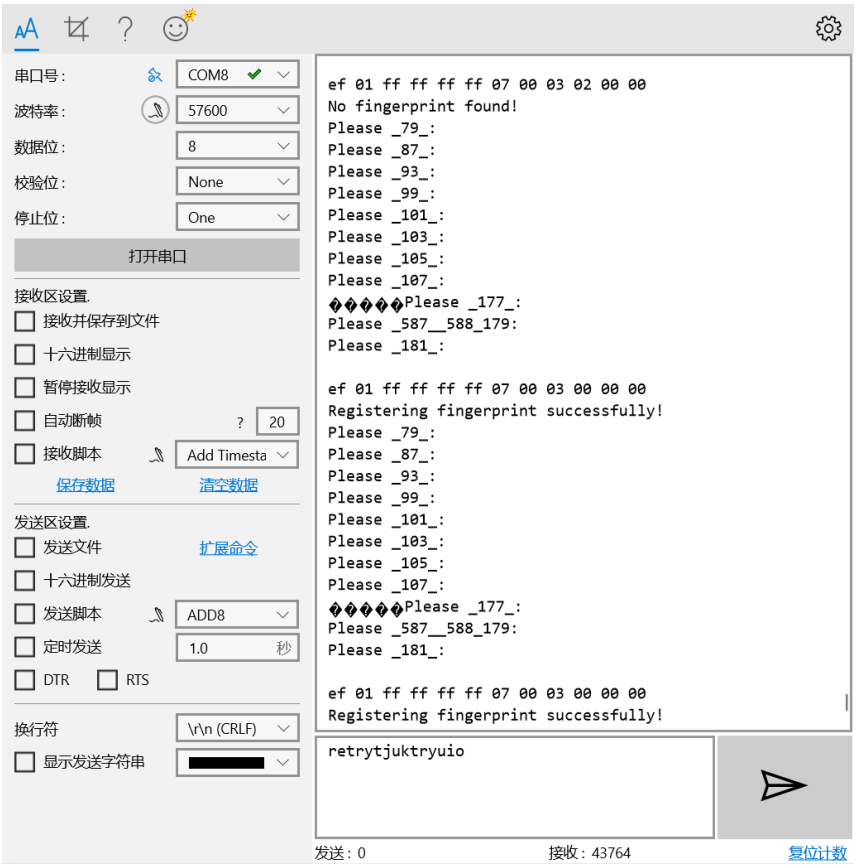


图 8-5 测试结果

最后，案例只是尝试了指纹模块的图像录入功能，实乃冰山一角，更多的功能测试还是需要用户去根据设备厂商给出的说明手册去尝试编写相应的程序去验证。

9. IIC 实验

9.1 IIC 简介

IIC（Inter-Integrated Circuit）其实是 IIC Bus 简称，中文应该叫集成电路总线，它是一种串行通信总线，使用多主从架构，由飞利浦公司在 1980 年代为了让主板、嵌入式系统或手机用以连接低速周边设备而发展。I2C 串行总线一般有两根信号线，一根是双向的数据线 SDA，另一根是时钟线 SCL。所有接到 I2C 总线设备上的串行数据 SDA 都接到总线的 SDA 上，各设备的时钟线 SCL 接到总线的 SCL 上。在 I2C 总线传输过程中，将两种特定的情况定义为开始和停止条件：当 SCL 保持“高”时，SDA 由“高”变为“低”为开始条件；当 SCL 保持“高”且 SDA 由“低”变为“高”时为停止条件。开始和停止条件均由主控制器产生。使用硬件接口可以很容易地检测到开始和停止条件。

9.2 LM75A 实验

LM75A 是一个高速 I2C 接口的温度传感器，可以在-55℃～+125℃的温度范围内将温度直接转换为数字信号，并可实现 0.125℃的精度。MCU 可以通过 I2C 总线直接读取其内部寄存器中的数据，并可通过 I2C 对 4 个数据寄存器进行操作，以设置成不同的工作模式。LM75A 有 3 个可选的逻辑地址管脚，使得同一总线上可同时连接 8 个器件而不发生地址冲突。为了辅助理解，下图给出了它的一种原理图（截取自 TI 公司的芯片数据手册）。

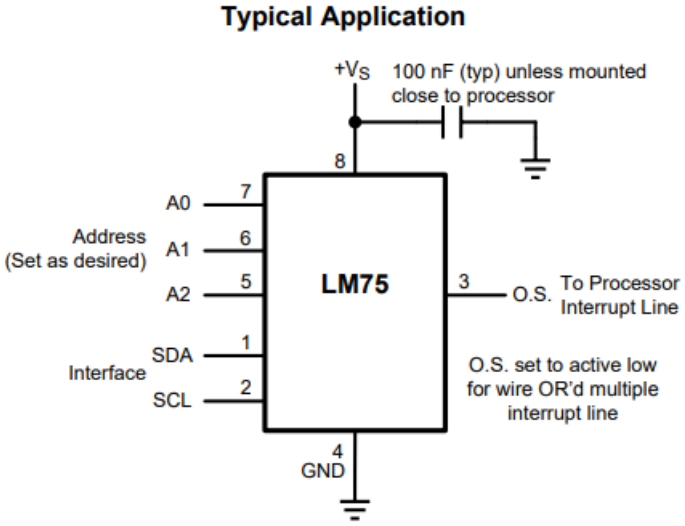


图 9-1 LM75A 引脚图

可以看到 A0 A1 A2 三个管脚可以标志芯片的逻辑地址，通过上拉或者下拉

不同位代表芯片的地址编号。LM75A 可配置成不同的工作模式。它可设置成在正常工作模式下周期性地对环境温度进行监控，或进入关断模式来将器件功耗降至最低。

在驱动 LM75A 读取环境的温度过程中，最重要的就是知道如何告诉芯片我要读取它的温度、它获得的温度存放在哪里以及我如何从存放温度的地方获取温度。这就引出了 LM75A 的寄存器的概念了。LM75A 温度寄存器是一个只读寄存器，包含 2 个 8 位的数据字节，由一个高数据字节（MS）和一个低数据字节（LS）组成。这两个字节中只有 11 位用来存放分辨率为 0.125℃的温度数据（以二进制补码数据的形式）。如下图所示，对于 8 位的 I2C 总线来说，只需从 LM75A 的温度寄存器地址（0x00）连续读两个字节即可（温度的高 8 位在前）。

7.6.2 Temperature Register (Read-Only):

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
MSB	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	LSB	X	X	X	X	X	X	X

D0–D6: Undefined. D7–D15: Temperature Data. One LSB = 0.5°C. Two's complement format.

图 9-2 LM75A 寄存器

其他配置寄存器则请查阅芯片的数据手册：

https://www.ti.com.cn/cn/lit/ds/symlink/lm75a.pdf?ts=1610087820340&ref_url=https%253A%252F%252Fwww.ti.com.cn%252Fproduct%252Fcn%252FLM75A

综上所述可以知道我们的目标就是告诉 LM75A 感受一下环境温度然后它就把温度放到温度寄存器中了，最后我们再读取温度寄存器里的内容，获取温度的工作基本就已经成功了。那么问题来了，如何跟 LM75A 握手通信呢？芯片数据手册中也有提到，如下图所示，片选和读写指令可以通过一个字节完成，规定为 1001_A0A1A2_R/W(高四位用 1001 标志为 LM75A 类型芯片，接下来就是前面提到的逻辑地址，根据你所购买或焊接的芯片具体情况具体配置，最低位为读写选择位)。

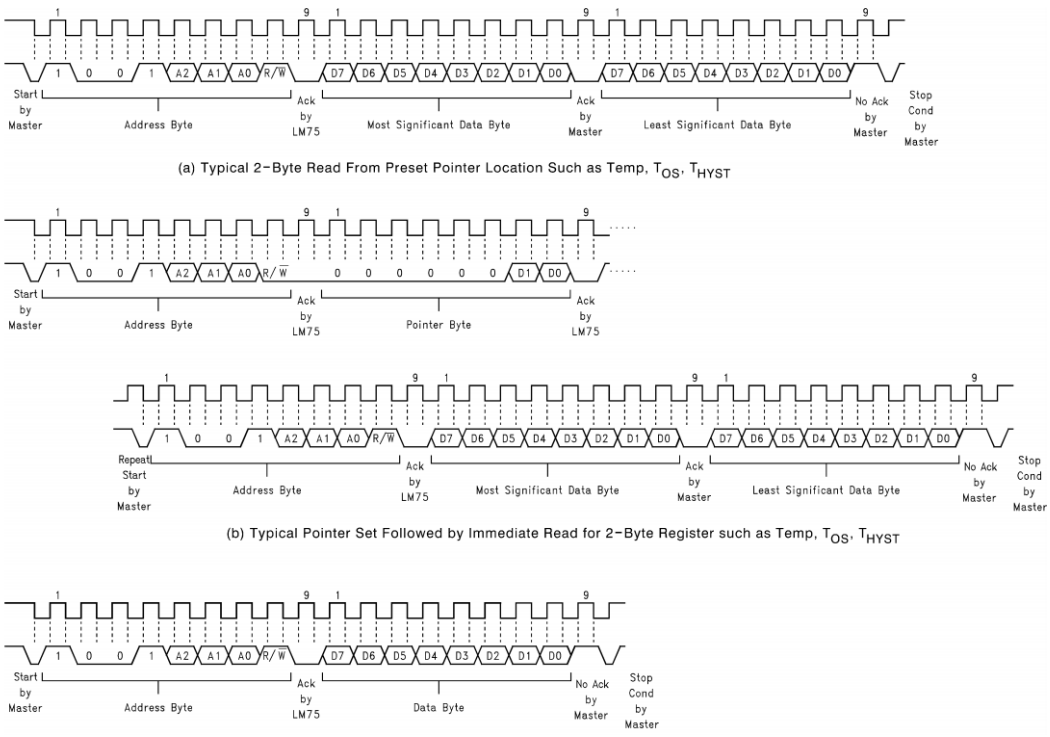


图 9-3 时序读写图

通过 IIC 写函数和 IIC 读函数去实现 LM75A 的驱动，需要注意的是，我们的 IIC 读函数是读取一个字节，所以读出来的温度精确度是 1°，如果想要连续全部读取温度寄存器的数据则需要用 DMA 搬运的方式连续读取两个字节的内容。

```
void Ro_IIC_LM75A(void)//20200601 ok
{
    uint8_t temp=0,i=0;
    IIC_Debug("_%s_:%d\r\n",__FUNCTION__,__LINE__);
    Ro_IIC_Write(0x90 ,0x00,1);
    temp=Ro_IIC_Read(0x91,1);
    IIC_Debug("temp is %d\r\n",temp);
}
```

如果测试基本 IIC 协议方式和 DMA 搬运下的温度读取，结果如图所示，可以明显发现 DMA 搬运方式下的温度精度位 0.125°，而普通 IIC 读取的温度只能读取到整数度。

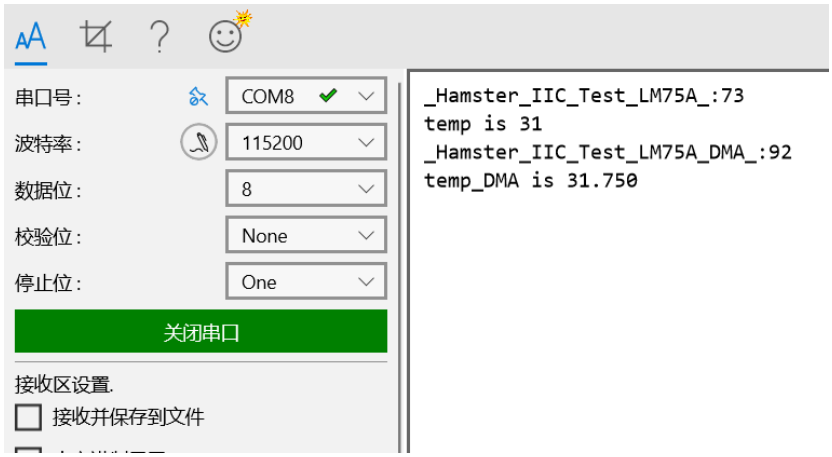


图 9-4 LM75A 读数据结果

9.3 AT24C02 实验

AT24C02 是一个 2K 位串行 CMOS E2PROM，内部含有 256 个 8 位字节。AT24C02 有一个 8 字节页写缓冲器。该器件通过 IIC 总线接口进行操作，有一个专门的写保护功能。AT24C02 支持 IIC，总线数据传送协议 IIC，总线协议规定任何将数据传送到总线的器件作为发送器。任何从总线接收数据的器件为接收器。数据传送是由产生串行时钟和所有起始停止信号的主器件控制的。主器件和从器件都可以作为发送器或接收器，但由主器件控制传送数据（发送或接收）的模式，由于 A0、A1 和 A2 可以组成 000~111 八种情况，即通过器件地址输入端 A0、A1 和 A2 可以实现将最多 8 个 AT24C02 器件连接到总线上，通过进行不同的配置进行选择器件。上可同时连接 8 个器件而不发生地址冲突。为了辅助理解，下图给出了我们测试用例驱动的 AT24C128 芯片的原理图（截取自微芯公司的芯片数据手册）。

Pin Configurations and Pinouts

Table 1-1. Pin Configuration

Pin	Function
A_0	Address Input
A_1	Address Input
A_2	Address Input
GND	Ground
SDA	Serial Data
SCL	Serial Clock Input
WP	Write Protect
V_{CC}	Device Power Supply

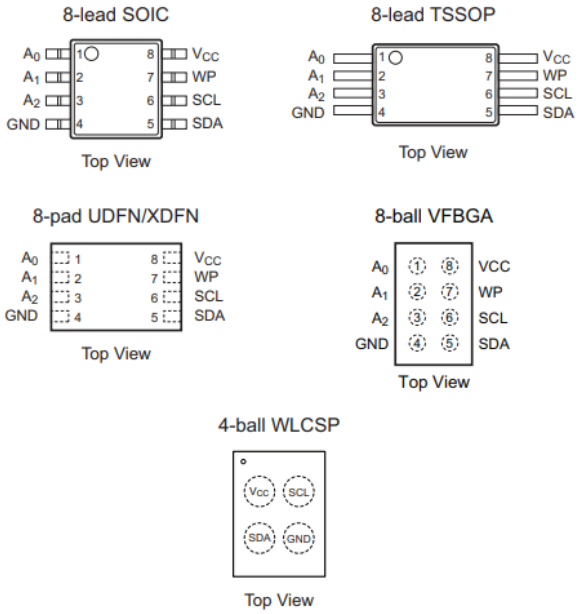


图 9-5 AT24C128 引脚定义

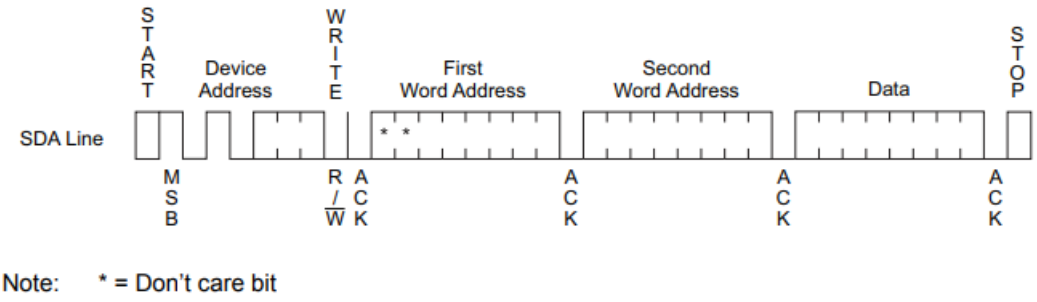
芯片的数据手册：http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8734-SEEPROM-AT24C128C-Datasheet.pdf

可以看到 A0 A1 A2 三只管脚可以标志芯片的逻辑地址，通过上拉或者下拉的不同标志唯一一块芯片。在驱动 AT24Cxx 芯片读写数据过程中，最重要的就是知道以什么样的数据格式告诉芯片我们要读取或者写入数据。在数据手册中，我们可以知道想要与 AT24Cxx 芯片握手通信，第一个字节发送的内容应该是 1010_A0A1A2_R/W 格式。

Package	Device Type Identifier				Hardware Slave Address Bits			Read/ Write
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SOIC, TSSOP, UDFN, XDFN, VFBGA	1	0	1	0	A_2	A_1	A_0	R/$\overline{W}$

图 9-6 AT24C128 握手格式

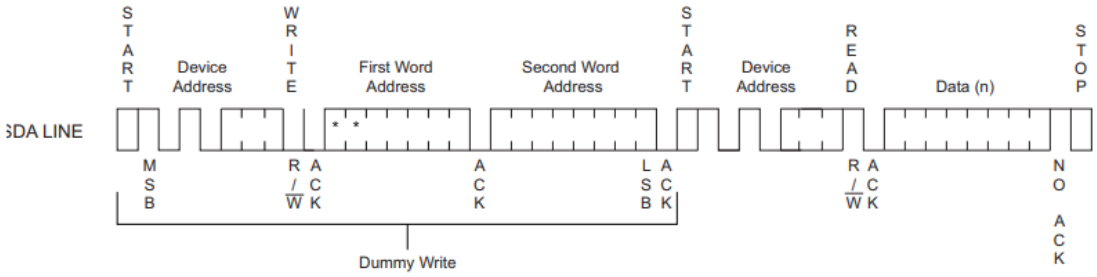
握手成功后官方继续给出了数据读写的标准格式，这里我们只介绍字节写入方式。如图所示，在 1010_A0A1A2_0 表明要写入数据之后，先写入两个字节的地址，随后写入想要写的数据，共三个字节。（这里可以看出如果多字节发送方式则只需要在发送完地址之后跟入想要发送的多个字节的数据即可）



Note: * = Don't care bit

图 9-7 AT24C128 写入格式

知道了数据写入方式之后我们来看看如何读取数据，因为我们能够控制数据写入 EEPROM 的某个位置，所以我们也要读取规定的位置，官方给出的读取方式如下图，先规定要读取的数据地址，再执行读取操作获取所存储的数据。



Note: * = Don't care bit

图 9-8 AT24C128 读出格式

综上所述我们驱动 AT24C128 芯片，如果你要给它发消息，那最低位就要置为 0（也就是发 1010_A0A1A20）；如果你要等它给你发消息了，最低为置为 1 即可（也就是发 1010_A0A1A21）。

9.4 案例设计

以 IIC 读写方式为例，通过 IIC 写函数和 IIC 读函数去实现 AT24C128 的驱动，需要注意的是，IIC 写函数是写入一个字节，而刚刚数据手册提到芯片需要两个字节的地址，所以如下图所示在 module_at24cxx.c 中我们用 DMA 搬运的方式实现对两个字节地址的写入任务。

```
void Ro_AT24CXX_Write_DMA(uint8_t *writedata,int len)
{
    DMA_HandleTypeDef DMA_RAM_IIC;
    Ro_Init_DMA_CLK();
    DMA_RAM_IIC.Instance=DMA0_Channel0;
    DMA_RAM_IIC.Init.Priority=DMA_PRIORITY_VERY_HIGH;
    DMA_RAM_IIC.Init.SrcType=DMA_DEFAULT;
```

```
    DMA_RAM_IIC.Init.DstType=DMA_IIC;
    DMA_RAM_IIC.Init.Mode=DMA_MODE_BYTE;
    Ro_Init_DMA(&DMA_RAM_IIC);
    Ro_DMA_Start(&DMA_RAM_IIC,(uint32_t)writedata,IIC_BASE_ADDR+IIC_TX_DATA
0,len);
}

/**
 * @brief AT24Cxx 读取单字节数据
 * @param ReadAddr:  需要读取的地址
 *
 * @retval uint8_t:  返回读取的数据
 */
uint8_t Ro_AT24CXX_ReadOneByte(uint16_t ReadAddr)
{
    uint8_t temp=0,rdmaaddr[2]={ReadAddr>>8,ReadAddr%256};

    Ro_IIC_Init();
    if(EE_TYPE>AT24C16)
    {
        Ro_IIC_Setaddr(0xA0,sizeof(rdmaaddr));
        Ro_AT24CXX_Write_DMA(rdmaaddr,sizeof(rdmaaddr));
    }
    else
    {
    }
    temp=Ro_IIC_Read(0xA1,1);
    return temp;
}
/**
 * @brief AT24Cxx 写单字节数据
 * @param WriteAddr:  写入数据的地址  DataToWrite:  欲写入的数据
 *
 * @retval None
 */
void Ro_AT24CXX_WriteOneByte(uint16_t WriteAddr,uint8_t DataToWrite)
{
    uint8_t wdmaaddr[2]={WriteAddr>>8,WriteAddr%256};
    Ro_IIC_Init();
    if(EE_TYPE>AT24C16)
    {
        printf("address is --- %x\r\n",WriteAddr);
        Ro_IIC_Setaddr(0xA0,sizeof(wdmaaddr));
```

```
        Ro_AT24CXX_Write_DMA(wdmaaddr,sizeof(wdmaaddr));
    }
    else
    {
    }
    Ro_IIC_Write(WriteAddr,DataToWrite,1);

}
```

案例是通过 DMA 搬运方式发送地址数据从而实现对 AT24C128 芯片的驱动，如下图所示，在总测试之前用 uint8_t Ro_AT24CXX_Check(void)函数提前测试芯片的单字节数据写入和读取工作，测试通过后进一步测试多字节字符串的读写操作。

如果阅读 void Ro_AT24CXX(void)函数内容我们可以发现如果单字节数据读写功能正常，串口助手会打印 24Cxx Ready!字样，随后即可开始读写字符串测试。结果如图所示，可以发现在 Ready 之后写入的字符串与读出的一致，测试通过。

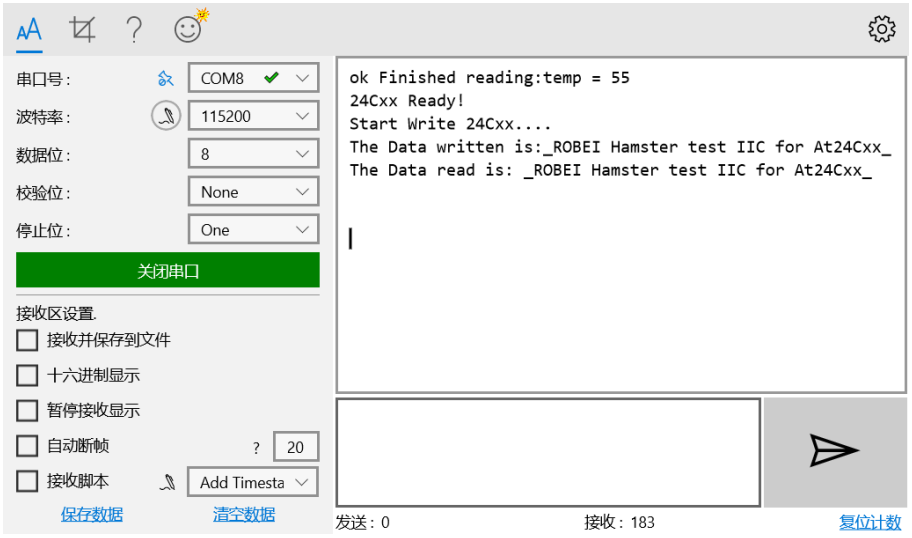


图 9-9 AT24C128 读写结果

9.5 GPIO 模拟 I2C

通过 GPIO 软件模拟来实现 IIC 的功能，如：IIC 起始信号产生/终止信号产生/产生应答信号/不产生应答信号/发送单字节数据/IIC 读取字节数据。IIC 函数意思是用到上面列举到的一个个功能（调用对应的函数），包含初始化->IIC 起始信号产生->IIC 终止信号产生->IIC 产生应答信号->IIC 不产生应答信号->IIC 发送单字节数据 0x55。IIC_Send_Byte(uint8_t b_data)函数会进行 8 次判断并打印对应结果->IIC 读取字节数据并且赋值给 temp，最后打印出来。

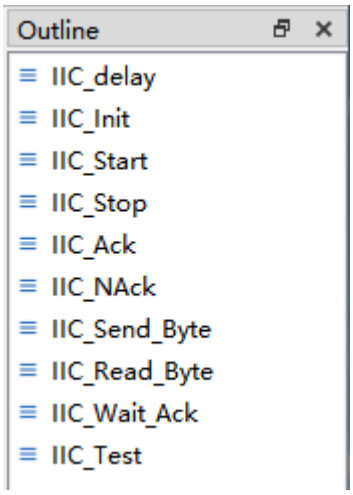


图 9-10 I2C 调用函数列表

```
void IIC(void)
{
    uint8_t Send_Data=0x55,temp=0x00;

    IIC_Init(I2CPORT);
    IIC_delay(500);
    IIC_Start();
    IIC_delay(500);
    IIC_Stop();
    IIC_delay(500);
    IIC_Ack();
    IIC_delay(500);
    IIC_NAck();
    IIC_delay(500);
    IIC_Send_Byte(Send_Data);
    IIC_delay(500);
    temp=IIC_Read_Byte(0);
    printf("Read data without Ack is %x\r\n",temp);
    IIC_delay(500);
    temp=IIC_Read_Byte(1);
    printf("Read data with Ack is %x\r\n",temp);
    while(1)
    {
    }
}
```

在 main 函数中调用 IIC 测试函数并 build 和编译即可：

```
int main( void )
{
    huart0.Instance=UART0;
```

```
    Ro_Init_Uart(&huart0);
    huart1.Instance=UART1;
    Ro_Init_Uart(&huart1);
    IIC();
}
```

结果如下：

图 9-11 软件模拟 I2C

10. SPI 实验

10.1 SPI TFT_LCD 实验

10.1.1 SPI 简介

本案例是针对规格为 3.5 寸的 TFT_LCD 屏（分辨率 320*480）制作的实验用例，显示屏支持 SPI 通信协议。SPI 是串行外设接口（Serial Peripheral Interface）的缩写，是一种高速的，全双工，同步的通信总线，并且在芯片的管脚上只占用四根线，节约了芯片的管脚，同时为 PCB 的布局上节省空间。SPI 总线是一种 4 线总线，因其硬件功能很强，所以与 SPI 有关的软件就相当简单，使中央处理器（Central Processing Unit，CPU）有更多的时间处理其他事务。SPI 是一种高速、高效率的串行接口技术。通常由一个主模块和一个或多个从模块组成，主模块选择一个从模块进行同步通信，从而完成数据的交换。SPI 是一个环形结构，通信时需要至少 4 根线（事实上在单向传输时 3 根线也可以）。SPI 的通信原理很简单，它以主从方式工作，这种模式通常有一个主设备和一个或多个从设备，需要至少 4 根线，事实上 3 根也可以（单向传输时）。也是所有基于 SPI 的设备共有的，它们是 MISO（主设备数据输入）、MOSI（主设备数据输出）、SCLK（时钟）、CS（片选）。

10.1.2 TFT LCD 软件设计

了解了 SPI 协议的基本概念之后，我们就来简单地看看我们的案例是如何用 SPI 协议驱动 LCD 屏幕的吧。如下图所示，在案例中，为了给显示文字、贴图等特殊图案做准备工作，特意准备了 module_LCDfont.h 和 module_pic.h 文件，存放了各种显示需要的素材。

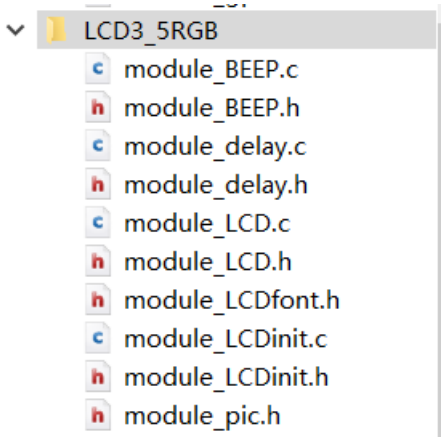


图 10-1 LCD 显示文件列表

此外，为了标志实验程序的启动，案例还设计了一路蜂鸣器来告诉使用者程序启动与否。细心观察购买的 LCD 屏幕，可以发现除了 SPI 通信协议中有的 CLK

和 MOSI 信号线外，还有屏幕配置相关的 RES、DC 和 BLK 信号线，这些就需要我们自行分配 GPIO 管脚去实现高低输出控制，管脚的数据控制就是 module_LCDinit.c 和 module_LCDinit.h 文件做的任务。最后，module_LCD.c 和 module_LCD.h 中存放我们对 LCD 控制的各种函数（如图）比如 LCD 显示字符和 LCD 画线条等等。

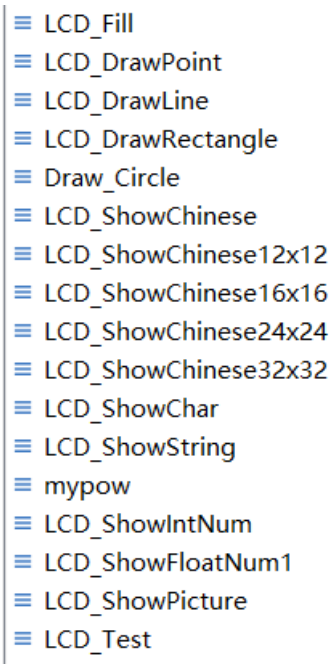


图 10-2 module_LCD 模块中的函数列表

案例代码：

//软件模拟的 CLK 和 MOS 要接 GPIO_PIN_0 和 GPIO_PIN_1；硬件 SPI 的 CLK 和 MOS 要接 GPIOD_PIN_6 和 GPIOD_PIN_1

```
void LCD (void)
    {
        hspi0.Instance=SPI0;
        hspi0.Init.CLKPolarity = SPI_POLARITY_HIGH;
        hspi0.Init.CLKPhase = SPI_PHASE_1EDGE;
        hspi0.Init.Mode = SPI_MODE_DATA ;
        Ro_SPI_Init(&hspi0);
        Ro_SPI_Cmd(&hspi0,0,0x01);
        MODIFY_REG(SPI0_REG(SPI_CTL0),DSS(0x1f),DSS(7)) ; //0=0x0a
        //WRITE_REG(SPI0_REG(SPI_CTL0),SPH(1)|DSS(7));
        LCD_Init();//LCD
        delay_lcd(50000);
        uint8_t i,j;
        float t=0;
        LCD_Fill(0,0,LCD_W-300,LCD_H,WHITE);
        LCD_Fill(LCD_W-300,0,LCD_W-200,LCD_H,RED);
        LCD_Fill(LCD_W-200,0,LCD_W-100,LCD_H,BLUE);
        LCD_Fill(LCD_W-100,0,LCD_W,LCD_H,GREEN);
```

```
        BEEP_Init();//BEEP
        BEEP_1;
        delay_lcd(5000);
        BEEP_0;

        while(1)
        {
            LCD_ShowChinese(0,0,"园子",RED,WHITE,32,0);
            LCD_ShowString(0,40,"LCD_W:",RED,WHITE,16,0);
            LCD_ShowIntNum(48,40,LCD_W,3,RED,WHITE,16);
            LCD_ShowString(80,40,"LCD_H:",RED,WHITE,16,0);
            LCD_ShowIntNum(128,40,LCD_H,3,RED,WHITE,16);
            LCD_ShowString(80,40,"LCD_H:",RED,WHITE,16,0);
            LCD_ShowString(0,70,"Increaseing Num:",RED,WHITE,16,0);
            LCD_ShowFloatNum1(128,70,t,4,RED,WHITE,16);
            t+=0.11;
            for(j=0;j<7;j++)
            {
                for(i=0;i<8;i++)
                {
                    LCD_ShowPicture(40*i,200+j*40,40,40,gImage_1);
                }
            }
        }
    }
```

先关注 while(1)之前的部分，最关键的就是四句 LCD_Fill 函数，这四句实现了将规定的区域点亮为我们想要的颜色的功能；while(1)中是显示汉字、显示字符串、显示数字和显示小数等功能。

运行结果如下，背景色是我们的 LCD_Fill 函数规定的，小企鹅是我们预先装在 module_pic.h 中的。测试流程直接调用 LCD()函数即可。

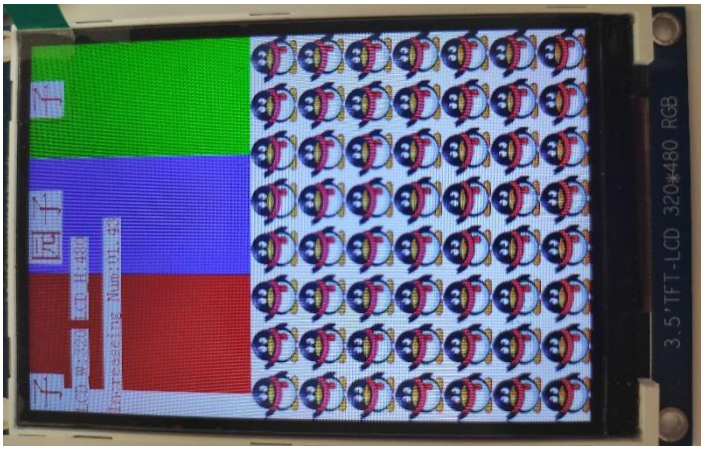


图 10-3 SPI 屏幕显示

10.2 Winbond flash 实验

10.2.1 SPI Flash 简介

本案例是针对华邦电子的 SPI Flash 产品编写的实验用例，因为是涉及到了具体电子公司的具体产品驱动测试，所以需要了解和准备的预备内容就自然比较多了。SPI Flash 就是一类以 SPI 协议工作的 Flash，可以实现数据的读写存储。不同的品牌（如华邦、旺宏和兆易等）生产的 Flash 芯片所规定的指令操作数也有不同程度的区别，本案例只针对华邦电子的 W25Qxx 系列芯片进行功能性实验，目的是打通 SPI 协议驱动 SPI Flash 的整个流程。

10.2.2 SPI Flash 软件设计

在案例中，module_winbond.c 一开头就给出了常用的操作指令（如图所示），例如写使能指令、状态读取指令、扇区擦除指令和页面编程指令等等。

```
#define WB_SINGLE_PROGRAM    (0x02)
#define WB_SINGLE_READ       (0x03)
#define WB_READ_STATUS       (0x05)
#define WB_WRITE_EN          (0x06)
#define WB_SECTOR_ERASE      (0x20)
#define WB_WRITE_STATUS      (0x31)
#define WB_QUAD_PROGRAM      (0x32)
#define WB_QUAD_READ         (0xEB)

#define WB_SING_DATA 0x5a5aa5a3
#define WB_QUAD_DATA 0xbcdeedcb



#define SINGLE_OFFSIZE 0
#define QUAD_OFFSIZE 64

uint8_t u8SingleSPI[20]={0};
uint8_t u8QuadSPI[20]={0};
uint8_t u8TransSingleSPI[]={"DMA Trans Sigl SPI"};
uint8_t u8TransQuadSPI[]={"DMA Trans Quad SPI"};
```

图 10-4 W25QXX 的指令

如果仔细阅读华邦官方给出的芯片数据手册：

https://www.winbond.com/hq/product/code-storage-flash-memory/serial-nor-flash/?__locale=zh&partNo=W25Q256JV，我们可以知道这款芯片的总体驱动流程为：

规定工作方式为 SINGLE 或者 QUAD （如果是 QUAD 则需用 Quad Enable 指令使能）-> 写使能+两个 Bytes 的 Dummy（0x00 0x00）-> 扇区擦除+擦除的起始地址+Dummy（0x00） -> 写使能+两个 Bytes 的 Dummy（0x00 0x00） ->页面编程+数据存放起始地址+要写入的数据 -> 读数据（附带读取的起始地址）

注意：每次发送指令数据都需通过状态读取指令来判断芯片是否就绪，尤其

是执行擦除指令和数据写入后，需要循环读取状态确保擦除完成和数据写入完成。

10.2.3 案例说明

1. 接口

打开 winbond 的工程，点击左边 FileSystem 的文件夹 Demo->Ro_sdk->drivers->Ro_SPI.c，这个是 SPI0/SPI1 的操作接口。看到右边的 Outline 栏：

Ro_SPI_Init(SPI_HandleTypeDef *hspi);是对（选择 SPI0/SPI1）初始化，里面包含了初始化 SPI 的时钟，调用 Ro_SPI_GPIO_Init();来初始化 GPIO。

Ro_SPI_Cmd(SPI_HandleTypeDef*hspi,uint8_t mode,uint32_t cmd)是让 SPI(写入命令，选择 SPI0/SPI1，选择模式 0 /1，选择命令)。

其余的：ReadData(读数据)；WriteData(写数据)；QuadEN(QUAD 指令使能)；ReadStatus(读 SPI 状态)；WaitForReady(等待完成)；SetupLen(数据长度选择)

显然，这里的 hspi0/hspi1 的作用是方便我们分别对 SPI0/SPI1 配置。

2. 测试模块

打开 winbond 的工程，点击左边 FileSystem 的文件夹 Demo->ModuleRoutines->WinBond->module_winbond.c,这个是 SPI 模块的测试用例（旺宏 MXIC 和华邦 Winbond）。同样看到右边的 Outline 栏(操作指令部分上面已经解释)：

这里的 extern 是要引用在 Ro_SPI.c 已经定义的变量 hspi0/hspi1，让我们在 module_winbond.c 可以使用。

Ro_SPI 是综合模块测试的函数，它包含了：

①SPI0/SPI1 的功能测试

②DMA 方式下的 SPI0/SPI1 功能测试

③DMA 中断方式下的 SPI0 功能测试

（并且每一个功能测试都有对工作方式 SINGLE/QUAD 的测试）

```
hspi1.Instance=SPI1;
hspi1.Init.CLKPolarity = SPI_POLARITY_HIGH;
hspi1.Init.CLKPhase = SPI_PHASE_1EDGE;
hspi1.Init.Mode = SPI_MODE_FLASH;
Ro_SPI_Init(&hspi1);
hspi0.Instance=SPI0;
hspi0.Init.CLKPolarity = SPI_POLARITY_HIGH;
hspi0.Init.CLKPhase = SPI_PHASE_1EDGE;
hspi0.Init.Mode = SPI_MODE_FLASH;
Ro_SPI_Init(&hspi0);
```

所以先初始化 SPI1 和 SPI0，然后调用函数 Ro_SPI();即可进行 Winbond 模块的测试。

例：Winbond 测试用例的第一条函数 Ro_SPI0_WinBond(0);括号写入为 0，则测试的模式为 SINGLE，打印 Single_Ro_SPI0_WinBond_:。

并且在使能，写数据擦数据以后。

Ro_SPI_WriteData(&hspi0,SINGLE_OFFSIZE,WB_SING_DATA);这个函数确定了数据的地址为 SINGLE_OFFSIZE 定义为 0，WB_SING_DATA 定义为 0x1a5aa5a3（可以在 module_winbond.h 中修改想写入的地址和数），这样写入了数据 0x1a5aa5a3。

```
if(isQuad)
{
    Ro_SPI_SetupLen(&hspi0,SPI_QUAD_WORD);
    Ro_SPI_Cmd(&hspi0,0,WB_QUAD_PROGRAM);
    Ro_SPI_WriteData(&hspi0,QUAD_OFFSIZE,WB_QUAD_DATA);
}
else
{
    Ro_SPI_SetupLen(&hspi0,SPI_SINGLE_WORD);
    Ro_SPI_Cmd(&hspi0,0,WB_SINGLE_PROGRAM);//0x02
    Ro_SPI_WriteData(&hspi0,SINGLE_OFFSIZE,WB_SING_DATA);
}
```

Temp=Ro_SPI_ReadData(&hspi0,SINGLE_OFFSIZE) ;这句话表示读了地址 SINGLE_OFFSIZE，并赋值给 temp。

最后打印写入的 WB_SING_DATA 和读到的 temp，看前后是否一致，一致则说明实验成功。

```
229     if(isQuad)
230         SPI_Debug("SPI0 Quad   Write :< %08x >,read :< %08x >\r\n",WB_QUAD_DATA,temp);
231     //printf("S0 Qdrd :< %08x >\r\n\r\n\r\n",temp);
232     else
233         SPI_Debug("SPI0 Sigl   Write :< %08x >,read :< %08x >\r\n",WB_SING_DATA,temp);
234     //printf("S0 Sgrd :< %08x >\r\n\r\n\r\n",temp);
235 }
```

```
Single_Hamster_SPI0_Test_WinBond_:79
SPI0 Sigl   Write :< 1a5aa5a3 >,read :< 1a5aa5a3 >
Quad_Hamster_SPI0_Test_WinBond_:49
SPI0 Quad   Write :< bcdeedcb >,read :< cccccccc >
```

图 10-4 检查输出与预期是否一致

11. 软中断实验

11.1 软中断简介

本案例实验 CPU 软件控制中断的功能，通过控制 IT_SW_INT0 寄存器的 5~23 位来实现对 IRQ5~23 的触发，其中 IRQ12 和 IRQ13 为 Bootloader 在使用，这里不做软中断实验。

沉芯系列芯片使用中断架构是由硬件来寻址，用户实现只需要启动相应的中断使能，填充对应的中断函数即可

中断使能（可自己实现）	中断处理函数（名称固定）	备注
Ro_IT_Enable_GPIO5	Ro_GPIO45_IRQHandle	GPIO 4 5 触发中断
Ro_IT_Enable_GPIO6	Ro_GPIO6_IRQHandle	GPIO 6 触发中断
Ro_IT_Enable_GPIO7	Ro_GPIO7_IRQHandle	GPIO 7 触发中断
Ro_IT_Enable_GPIO0	Ro_GPIO0_IRQHandle	GPIO 0 触发中断
Ro_IT_Enable_GPIO1	Ro_GPIO1_IRQHandle	GPIO 1 触发中断
Ro_IT_Enable_GPIO2	Ro_GPIO2_IRQHandle	GPIO 2 触发中断
Ro_IT_Enable_GPIO3	Ro_GPIO3_IRQHandle	GPIO 3 触发中断
Ro_IT_Enable_DMA01_CH2	Ro_DMA01_CH2_IRQHandle	DMA0 和 DMA1 的 channel 2 共用此中断
Ro_IT_Enable_DMA01_CH3	Ro_DMA01_CH3_IRQHandle	DMA0 和 DMA1 的 channel 3 共用此中断
Ro_IT_Enable_DMA01_CH4	Ro_DMA01_CH4_IRQHandle	DMA0 和 DMA1 的 channel 4 共用此中断
Ro_IT_Enable_DMA01_CH5	Ro_DMA01_CH5_IRQHandle	DMA0 和 DMA1 的 channel 5 共用此中断
Ro_IT_Enable_DMA01_CH6	Ro_DMA01_CH6_IRQHandle	DMA0 和 DMA1 的 channel 6 共用此中断
Ro_IT_Enable_TM16	Ro_TIM16_IRQHandle	TIM1 TIM2 和 TIM3 共用一个中断向量
Ro_IT_Enable_TM32	Ro_TIM32_IRQHandle	TIM0 和 TIM4 共用一个中断向量
Ro_IT_Enable_DMA01_CH7	Ro_DMA01_CH7_IRQHandle	DMA0、1 的 channel 7 共用此中断
Ro_IT_Enable_ADP	Ro_Adptive_IRQHandle	Adaptive 触发的中断
初始化中打开使能	Ro_UART0_IRQHandle	UART0 触发中断
初始化中打开使能	Ro_UART1_IRQHandle	UART1 触发中断
初始化中打开使能	Ro_IIC_IRQHandle	IIC 触发的中断
默认使能	Ro_Crash_IRQHandle	系统崩溃的中断

11.2 软中断软件设计

在 Demo->robei_sdk->drivers->robei_IT.c 中找到 IRQ5~23 的使能函数以备用，发现没有关于 IRQ23 的使能，这是因为我们在 CPU 设计时将没用到的中断规定为默认配置后就没有给它们写接口了，这样的话我们在 module_softIT.c 中照猫画虎自己写了一个使能函数如下图：

```
/**
  * @brief  使能 IRQ23
  * @retval None
  */
void Ro_IT_Enable_IRQ23(void)
{
    SET_BIT(PLIC_REG(IT_MASK),IT_IRQ23) ;
    MODIFY_REG(PLIC_REG(IT_REQA23),EX(0xf),EX(0xe));
}
```

这样就可以完整地写出软中断使能函数，完成后是这个样子：

```
/**
  * @brief  软中断的所有中断使能
  * @retval None
  */
void Ro_SoftIT_ITEN(void)
{
    Ro_IT_Enable_GPIO5();//5
    Ro_IT_Enable_GPIO6();//6
    Ro_IT_Enable_GPIO7();//7
    Ro_IT_Enable_GPIO0();//8
    Ro_IT_Enable_GPIO1();//9
    Ro_IT_Enable_GPIO2();//10
    Ro_IT_Enable_GPIO3();//11

    Ro_IT_Enable_DMA01_CH2();//14
    Ro_IT_Enable_DMA01_CH3();//15
    Ro_IT_Enable_DMA01_CH4();//16
    Ro_IT_Enable_DMA01_CH5();//17
    Ro_IT_Enable_DMA01_CH6();//18
    Ro_IT_Enable_TM16();//19
    Ro_IT_Enable_TM32();//20
    Ro_IT_Enable_DMA01_CH7();//21
    Ro_IT_Enable_ADP();//22
    Ro_IT_Enable_IRQ23();//23
```

```
}
```

使能好中断之后它们就处于待命状态了，如果我们拉高 IT_SW_INT0 寄存器的第 5 位，那么 IRQ5 中断就会被触发；拉高第 15 位，那么 IRQ15 中断就会被触发，以此类推。如图所示就是我们依次拉高这些中断的触发标志位的函数：

```
/**
  * @brief  软中断的所有中断按顺序触发
  * @retval None
  */
void Ro_SoftIT_Trig(void)
{
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ5) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ6) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ7) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ8) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ9) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ10) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ11) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ14) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ15) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ16) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ17) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ18) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ19) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ20) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ21) ;
    Delay_softIT(500);
```

```
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ22) ;
    Delay_softIT(500);
    SET_BIT(PLIC_REG(IT_SW_INT0),IT_IRQ23) ;
    Delay_softIT(500);
    }
```

有了中断使能并且触发了中断之后，我们如何知道触发成功与否呢？这就需要我把中断服务程序按照中断编号编写出好辩认的内容呈现在串口调试助手上。例如 IRQ5_Handle()中用几句 uart0_putchar()去打印 5OK 字样，IRQ6_Handle()中用几句 uart0_putchar()去打印 6OK 字样。如图所示：

```
/**
  * @brief   IRQ5 中断服务函数
  * @retval None
  */
void Ro_GPIO45_IRQHandle(void)
{
     uart0_putchar('5');
     uart0_putchar('O');
     uart0_putchar('K');
    uart0_putchar('\n');
}
/**
  * @brief   IRQ6 中断服务函数
  * @retval None
  */
void Ro_GPIO6_IRQHandle(void)
{
    uart0_putchar('6');
    uart0_putchar('O');
   uart0_putchar('K');
    uart0_putchar('\n');
}
```

写完 5~23 之间的所有服务函数之后，在 void Ro_SWIT(void)函数中先调用使能函数，再调用依次触发中断的函数，最后在 main 函数中调用它就能观察结果。

```
/**
  * @brief   软中断测试
  * @retval None
  */
void Ro_SWIT(void)
{
    Ro_SoftIT_ITEN();
    Ro_SoftIT_Trig();
    while(1);
```

```
}
int main( void )
{
    huart0.Instance=UART0;
    Ro_Init_Uart(&huart0);
    huart1.Instance=UART1;
    Ro_Init_Uart(&huart1);
    Ro_SWIT();
}
```

结果如下图所示：

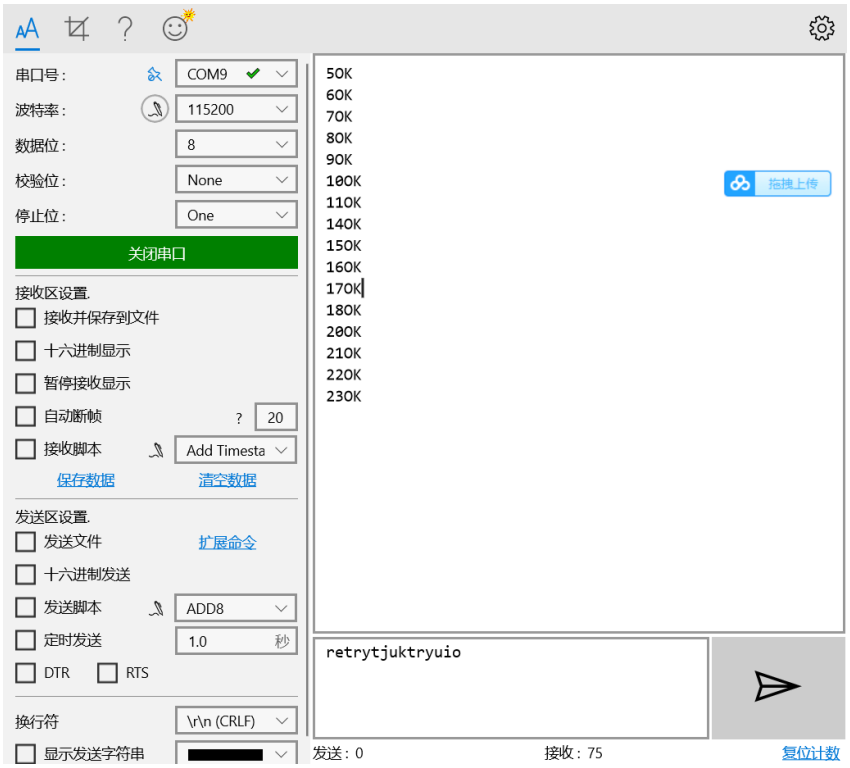


图 10-5 软中断案例运行结果

12. RTOS 案例

12.1 FreeRTOS 简介

FreeRTOS 是一个迷你实时操作系统的内核。作为一个轻量级操作系统，功能包括：任务管理、时间管理、信号量、消息队列、内存管理、记录功能、软件定时、协程等，可基本满足较小系统的需要。

由于 RTOS 需占用一定的系统资源(尤其是 RAM 资源)，只有 μC/OS-II、embOS、salvo、FreeRTOS 等少数实时操作系统能在小 RAM 单片机上运行。相对 μC/OS-II、embOS 等商业操作系统，FreeRTOS 操作系统是完全免费的操作系统，具有源码公开、可移植、可裁减、调度策略灵活的特点，可以方便地移植到各种单片机上运行

12.2 FreeRTOS 软件设计

现有的工程中已经移植 FreeRTOS，如下图：

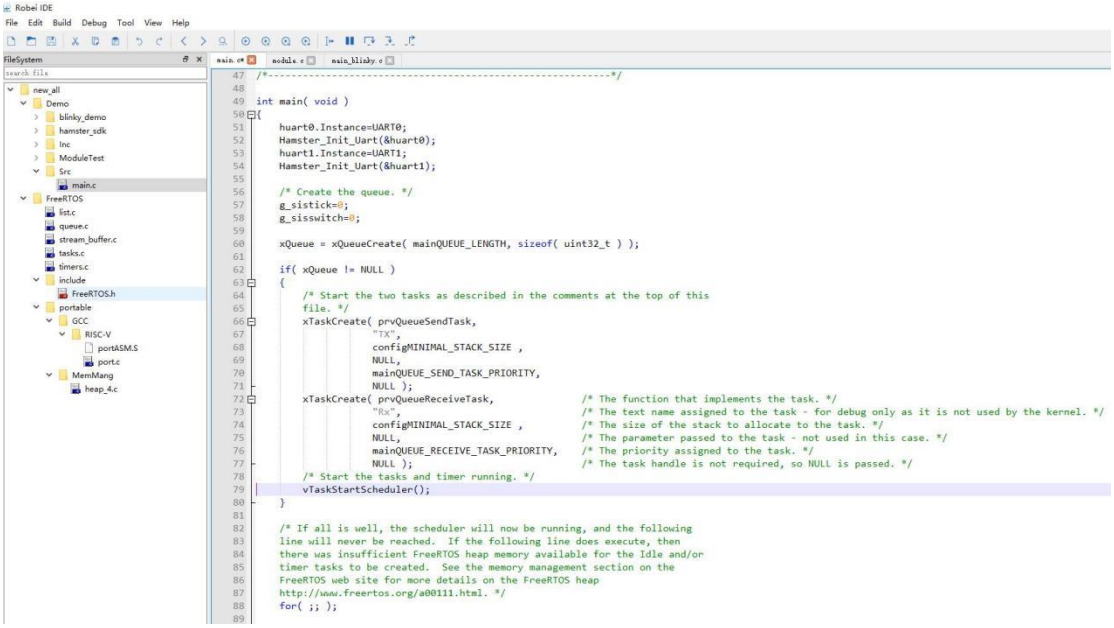


图 12-1 FreeRTOS 工程案例

在 main 函数中创建两个线程，线程 1 通过 UART0 输出'_'，线程 2 通过 UART0 输出'|'

```
int main( void )
{
    huart0.Instance=UART0;
```

```
    Ro_Init_Uart(&huart0);
    huart1.Instance=UART1;
    Ro_Init_Uart(&huart1);

    /* Create the queue. */
    g_sistick=0;
    g_sisswitch=0;

    xQueue = xQueueCreate( mainQUEUE_LENGTH, sizeof( uint32_t ) );

    if( xQueue != NULL )
    {
   /* Start the two tasks as described in the comments at the top of this ile. */
        xTaskCreate( prvQueueSendTask,
                     "TX",
                     configMINIMAL_STACK_SIZE ,
                     NULL,
                     mainQUEUE_SEND_TASK_PRIORITY,
                     NULL );
        xTaskCreate( prvQueueReceiveTask,                    /
                         "Rx",
                         configMINIMAL_STACK_SIZE ,
                         NULL,
                         mainQUEUE_RECEIVE_TASK_PRIORITY,
                         NULL );
            /* Start the tasks and timer running. */
            vTaskStartScheduler();
        }

        /* If all is well, the scheduler will now be running, and the following
        line will never be reached. */
        for( ;; );
    }

    static void prvQueueSendTask( void *pvParameters )
    {
        uint32_t i=0;
        for( ;; )
```

```
        {
            if(i==2000)
            {
                uart0_putchar('_');
                i=0;
            }
            i++;
        }
    }

    static void prvQueueReceiveTask( void *pvParameters )
    {
        uint32_t i=0;
        for( ;; )
        {
            if(i==2000)
            {
                uart0_putchar('|');
                i=0;
            }
            i++;
        }
    }
```

实验结果：MCU 通过串口持续输出“_____|||||||||||||||_____|||||||||||||||”至电脑端。

13. JTAG 案例

13.1 Robei Link 模块简介

Robei Link 模块集成烧录和 debug 功能,win7 和 win10 系统可以直接使用，不需要额外安装驱动

图 13-1 Robei Link 模块

13.2 Robei Link 模块使用

Robei Link 连接电脑和开发板之后，可以使用 IDE 直接进行烧录和 Debug

- 编译下载
  - 工程属性。左边栏工程根节点的右键菜单中的 Properties 动作可以触发设置工程属性的对话框，用于设置工程编译的脚本、库包含目录等属性。
- 编译:分为编译、重编译、清除。
  - 错误查看。执行编译后，底部的 Compile Output 输出窗口会自动弹出，显示编译的信息，并且可以通过此窗口右上角的清除、关

闭按钮清空日志、中断编译进程。如果要查看编译的具体警告和错误，可以点击查看底边栏的 GCC report 输出窗口。

下载对话框：通过动作 Build -> Download Dialog... 可以打开下载对话框；对话框打开时会自动检查 Flash 连接是否正常，也可以通过 Refresh 按键 手动刷新 连接；当前工程的二进制文件的文件路径会自动加载到文件框上，也可通过 choose 按键，触发对话框、选择指定文件；通过点击对话框最顶层的 Read 按键、Erase 按键、 Program 按键对当前打开的 Flash 进行操作，Flash 的读取结果显示在中间输出去，最底下的状态栏显示当前操作的运行结果；烧写完成后，在电路板上按下重启按键，程序将会被激活。

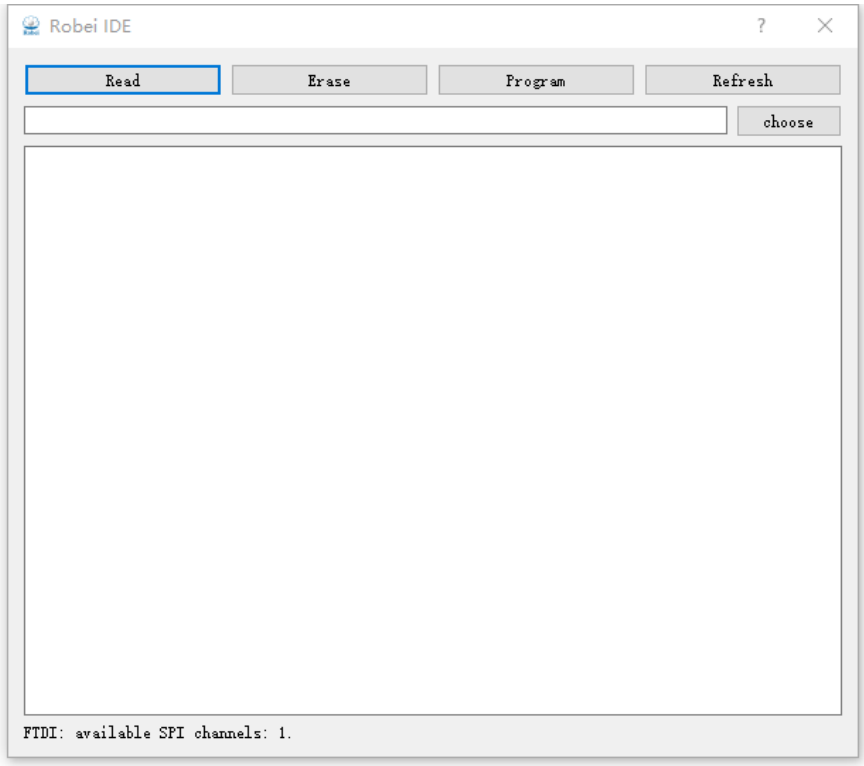


图 13-2 Robei IDE 下载窗口

一键下载：在电路板正常连接的前提下，通过动作 Build -> Download 或工具栏的相应图标，可以实现一键下载。需要注意的是，在使用一键下载期间，不能打开下载对话框。

在线调试

- 添加、删除断点。编辑区行号的左边区域，通过左键点击，可以添加、删除断点。

- 输出窗口和出错信息。调试模式下，右边栏显示寄存器、断点、栈帧等各项信息，底边栏的 Debug Output 输出窗口显示 gdb 出错信息。

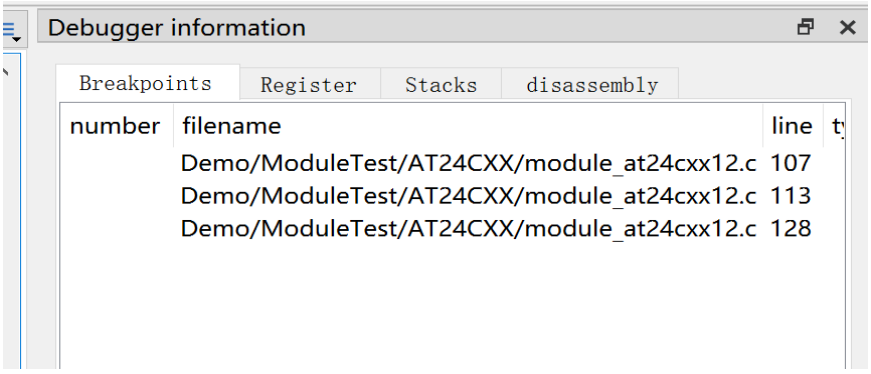


图 13-2 Debug 窗口信息

进入调试前要先设置好断点，通过菜单 debug -> Start Debugger 开始调试过程；调试器每次暂定在断点后，编辑区自动打开其断点对应的文件和行号，右边栏信息输出窗口会自动更新此时的各项参数；通过菜单 debug -> continue 或工具栏的对应按钮，运行到下一个断点；通过菜单的 debug -> Step Over，Step Into，Step Out 进行语句级别的调试、函数级别的调试； 通过 debug -> Pause 暂停调试，可以用于循环语句、死循环中；通过 debug -> Stop 停止调试模式； 通过 debug->Force Stop 强制终止调试。

13. 自适应处理器

13.1 自适应架构概述

当集成电路的制程工艺越来越接近原子级别，新的芯片制造费用越来越高，硅工艺快走到极限的时候，还有什么办法来满足社会上不断增长的计算量需求？以 Intel 和 ARM 为代表的公司采用多核并行执行来提升计算能力，但是随着核数增加，核与核之间数据的等待，既增加了能耗，也导致真正计算性能远远低于预期，尤其是功耗成了移动计算平台的头号难题。高性能与低功耗是芯片设计的两个相悖方向，所有计算芯片公司都在努力寻求一个高性能低功耗的解决方案。如 Intel 通过兼并 Altera， AMD 兼并 Xilinx，都是想通过 CPU+FPGA 异构的方式来实现高性能低功耗的目的，这种方案无疑加大了项目设计难度，因为 FPGA 和 CPU 编程与原理各不相同，既使可以通过高级语言综合将 C/C++/SystemC 转化成 RTL，也很难在软硬件结合的环境中准确控制其时序，这种方案要求设计人员精通完全不同的软件与硬件设计模式，但是这两套设计模式相冲突，普通人很难灵活切换与掌控。随着技术的发展，片上网络（NOC：Network on chip）架构也逐渐进入产业界，也就是一颗芯片内部集成成百上千的精简化的 RISC 处理器核心，然后核与核之间通过特制网络协议进行通讯。当大量数据涌入的时候，每个核都要将传输到自己位置上的包识别一次，如果确认该包属于自己，则执行数据，否则转发到下一个核。海量数据在片上多核网络芯片的网络上传输，会大大降低数据到达时间，增加延迟与功耗，每个核等待时间和包识别时间都是功耗上升的关键因素。

除了常见的芯片架构外，还有一些特殊架构也在不断兴起，首先是不做芯片的 Google（Alphabet）开始入手设计芯片，并推出了 Tensor 深度学习芯片，之后又有 IBM 推出的模拟人类大脑的神经元芯片 TrueNorth，该类芯片将深度学习神经网络做成芯片，用来做人工智能和图像识别以及大数据处理等。但是神经网络不是采用传统编程模式来设计的，而是通过海量数据训练来达到预期效果，所以，在使用上，要采用数据训练的方式进行应用，比传统芯片开发周期长。海量数据获取与样本鉴别又是一个巨大的挑战。另外，TrueNorth 芯片的运行频率受限制，计算带宽自然无法获得显著提升。

Robei 沉芯芯片架构是基于吴国盛提出的时空转换理论，是一种超越摩尔定律的集成电路设计方案。随着摩尔定律的失效，计算性能的提升无法再依赖于工艺提升。传统的 SOC 架构注定芯片设计走向垄断，虽然 SOC 集成了很多 IP，但是并不是每个 IP 同时都在使用，IP 模块是通过关断与切换来实现应用的，比如一开始在使用 A 模块，但是程序要运行 B 模块，就需要把 A 模块关掉，给 B 模块上电，重置一下寄存器，然后再使用 B 模块。关断与切换的过程大概耗时 100ns 到 1us 的时间，而且关断与切换的方式需要同时把 A 和 B 模块预先集成到硅晶圆上，会增加硅晶圆面积，也增加了功耗。沉芯异构芯片采用另外一种模式，采用高速动态可重构的计算阵列重构的方式实现，功能模块可以通过软件配置，不

需要预先集成，减少了晶圆面积和降低了功耗。沉芯自适应芯片可以通过不断的重构，来实现同一芯片上可以配置无限的功能，一眨眼可以重配置 10 万次以上，让用户感觉不到切换的时间，反而感觉到芯片需要什么就可以用什么功能，实现感觉上的芯片硅面积无限大。

图 13-1 时空转换理论

沉芯芯片通过软件定义芯片的模式打造出一种全新的架构，实现串并行无缝切换，传统 CPU/MCU 架构负责串行数据处理，自适应动态可重构阵列处理器负责数据并行处理，两个处理器之间采用 DMA 进行数据交换。沉芯芯片是按需配置的芯片，根据不同的时刻对硬件不同要求，进行调用不同的配置，达到微秒到纳秒级别的重构。当数据不需要并行处理的时候，传统 CPU 或 MCU 架构完全满足串行处理的需求，并行部分可以处于待机状态，节约能耗。当需要采用并行加速的时候，启用自适应并行处器，采用先配置后使用的方式来实现软件定义芯片。在运行过程中，传统架构可以随时响应自适应处理器的中断请求，实现实时动态可重构。沉芯打造出数据流与控制流的一体化，实现了软件定义芯片。数据在自适应处理器中可以多重复制与数据交叉，降低了反复存取数据带来的功耗的消耗。沉芯芯片适合在很多对计算性能有比较高要求同时功耗受限的场合，为客户提供最优化的处理方案。该芯片不仅内部结构可以软件重构，外部引脚也可以软件重构，用户的程序运行不需要重写程序，只需要提前配置引脚重映射就可以继续执行。

图 13-2 沉芯异构芯片

13.2 自适应阵列处理器

自适应阵列处理器由 Memio，Busmatrix，Rocell 阵列组成。Memio 是由行一行的 32 位存储器所构成，既可以作为输出，也可以作为输入，具体长度依据不同的沉芯芯片的型号来定。Busmatrix 作为数据走向的控制，用于链接 Rocell 的具体行与 Memio 的具体行数据。通过 Busmatrix 可以实现任一一行的 Memio 的数据传输到任一一行的 Rocell 阵列种。Rocell 阵列是由一堆异构的 Rocell 单元进行二维排布实现的，每个 Rocell 单元都可以与相邻的 8 个 Rocell 进行通信。

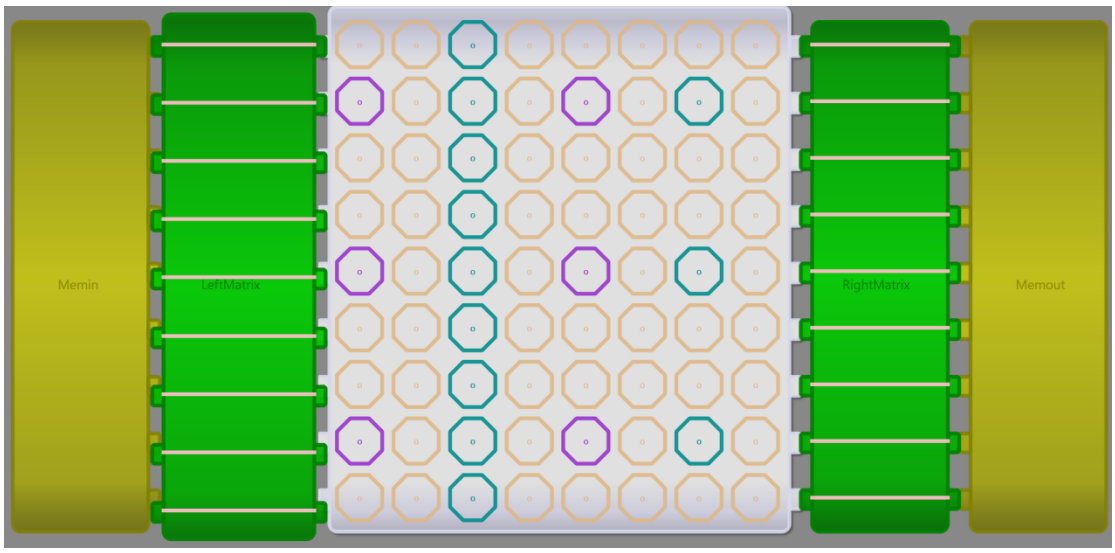


图 13-3 自适应阵列处理器结构

Rocell 阵列中每个 Rocell 只执行一种计算，将计算结果直接传递给相邻 Rocell 进行下一步计算。整个阵列中可以只使用部分 Rocell，未被使用的 Rocell 的时钟

被关断，不会有动态功耗，只有漏电功耗。由于采用了多个 Rocell 同时运算，将基于指令的计算方式打破，采用更深流水线级别实现并行加速，在维持计算性能不变的情况下，平行处理器阵列可以有效降低整个系统的运行频率，进而降低功耗。

13.2.1 Rocell 单元

Rocell 是平行处理器阵列的基本组成单元，根据功能不同，可以划分为三类：仅负责单周期运算的基础计算模块 Roce，带有浮点加减运算的 Rofl 和乘法器 Romul。Roce 含有 32 种基本的指令操作，Robu 在 Roce 的基础上增加了 4 条专用指令，Romul 在 Roce 的基础上增加了 2 条专用指令。每个 Rocell 根据结构划分为两个部分：输入输出控制器和计算核心 Rocal。Rocal 的计算架构如公式 1 所示，

$$\{\mathrm{co},\mathrm{x}\} = \mathrm{OP}(\mathrm{cin}_{selc}, \mathrm{b}_{selb}, \mathrm{a}_{sela}) \tag{1}$$

其中，co,x 为 Rocal 的输出端，OP 为运算函数的索引，$\mathrm{cin}_{selc}, \mathrm{b}_{selb}, \mathrm{a}_{sela}$为被选择的 Rocal 输入端。

$$\begin{cases} \mathrm{cin}_{selc} = \sum_{i=0}^{8} cin_i * (selc == i) \\ \mathrm{b}_{selb} = \sum_{i=0}^{8} b_i * (selb == i) \\ \mathrm{a}_{sela} = \sum_{i=0}^{8} a_i * (sela == i) \end{cases} \tag{2}$$

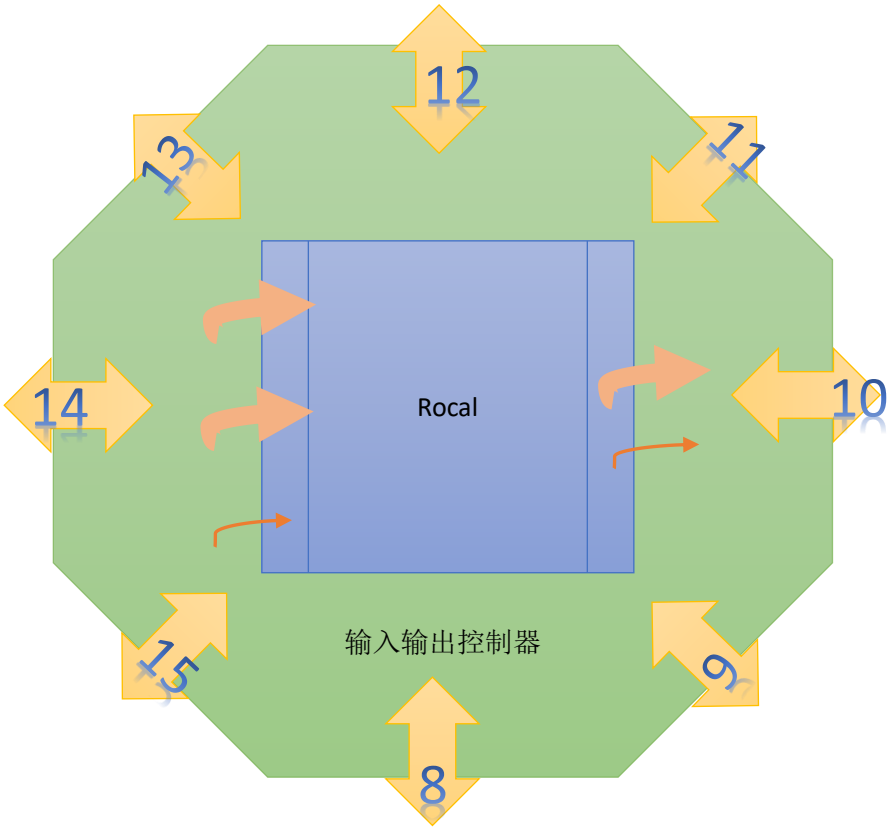


图 13-4 Rocell 基本处理单元

Roce 是 Rocell 中仅包含单周期运算指令的最小单元，其运算指令可以在一个时钟周期内完成。Romul 是包含浮点运算指令的计算单元，在 Roce 基础上，多出了 2 条计算指令，整型乘法和浮点乘法。Rofl 也是如此，但是多出来的 4 条指令为浮点与整型的转换、浮点加和浮点减。

13.3 自适应指令集

Rocell 指令集的归类按照 Rocell 的种类不同进行划分，Roce 指令涵盖 32 条基本指令，Rofl 和 Romul 拥有特殊指令单独介绍。

表 13-1 Roce、Rofl、Romul 共有指令集

编号	指令名	符号	功能	描述
00	Nop	o	空运算	不做任何事情，该单元不启用
01	Pass	->	传递	将数据传递到下一个单元
02	And	&	与运算	实现按位与操作
03	Or	|	或运算	实现按位或操作
04	Xor	^	异或运算	实现按位异或操作
05	Cmp	<>	与 0 比大小	X[0]=a>0;

				X[1]=a<0; X[2]=(a!=0)
06	Lsl	<<	逻辑左移	逻辑左移，co 也参与移位
07	Lsr	>>	逻辑右移	逻辑右移，co 也参与移位
08	Inc	++	自加运算	cin 为 1 重置起始值，cin 为 0 开始自加运算
09	Absi	|i|	整型数取绝对值	对整型数据取绝对值
10	Negi	~	整型数取反	对整型数取反
11	Merge	>=	双路合并	当两路数据都有效时，a 取低 16 位，b 取高 16 位
12	Addu	u+	无符号加法	无符号的加法运算，cin 和 co 都参与 {co,x}=a+b+cin
13	Subu	u-	无符号减法	无符号的减法运算，cin 和 co 都参与 {co,x}=a-b-cin
14	Adds	s+	有符号加法	x=a+b
15	Const	C	常数	输出常数 a 的值，a 可以存储到 Rocell 中
16	Sgn	sgn	获取符号位	获取符号位数据，输出为 co
17	Mux	=|-	两路数据选通	数据选择器，x=cin?b:a
18	Nand	~&	Nand 运算	按位 Nand 运算
19	Nor	~|	Nor 运算	按位 Nor 运算
20	Xnor	~^	Xnor 运算	按位 xnor 运算
21	Zero	z	判断为 0	如果输入为 0，co 输出 1，否则为 0
22	Rol	@	循环左移	co 参与的循环左移运算
23	Asr	<<<	算术右移	带符号位的算术右移
24	Dec	--	递减运算	cin 为 1 重置起始值，cin 为 0 开始自加运算
25	Absf	|f|	浮点数绝对值	对浮点数取绝对值操作
26	Negf	-f	浮点取反	对浮点数取反操作
27	Lut	lut	查找运算	co 输出在 a 中查找位置为 b 低 5 位的比特值
28	Carry	<co	进位运算	co 输出加法运算的进位
29	Borrow	bo>	借位运算	co 输出减法运算的借位
30	Subs	s-	有符号减法	有符号减法运算
31	Equal	==	判断相等	判断两个数是否相等

表 13-2 Rofl 独有指令集

编号	符号	符号	功能	描述
33	F2I	f=i	浮点转整型	按 IEEE-754 格式浮点转整型
34	I2F	i=f	整型转浮点	按 IEEE-754 格式整型转浮点

35	Addf	f+	浮点加法	浮点数的加法运算
36	Subf	f-	浮点减法	浮点数的减法运算

表 13-3 Romul 独有指令集

编号	符号	符号	功能	描述
37	Muli	i*	整型乘法	整型乘法器
38	Mulf	f*	浮点乘法	IEEE-754 标准浮点乘法

13.4 自适应开发工具

在 Robei IDE 的菜单中选择 Tool 菜单的下拉菜单 Adaptive，就可以打开 Adaptive IDE。Robei IDE 实现的是对 CPU 程序的编写与编译，Adaptive IDE 实现的是针对自适应处理器的配置与重构。

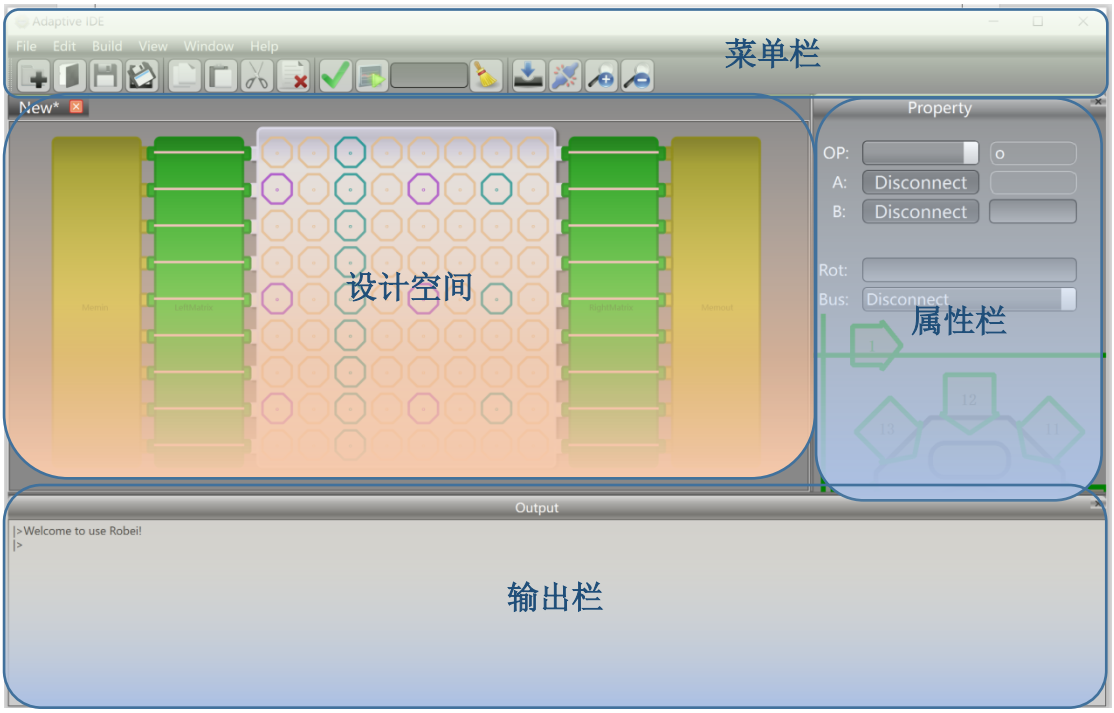


图 13-5 Adaptive IDE 工具界面

Adaptive IDE 是 Robei 旗下的自适应处理器可视化开发工具，该工具集合了 Rocell 配置、数据通路设定、数据对齐检查、单步执行仿真、一键仿真、生成配置文件、配置文件融合等功能，可以作为自适应处理器的可视化配置设计工具。该开发工具不需要太多的编程技能，界面可视化设计如同下棋一样，熟悉下棋规则，任何人都可以设计。Adaptive IDE 由菜单栏，属性栏、设计空间和输出窗口组成。菜单栏包含下拉菜单和工具栏，属性栏既可以配置 Rocell，又可以配置 Busmatrix。输出栏会将运行中的信息进行展示。

13.4.1 菜单栏

图 13-6 菜单栏

菜单栏顶部是常见的下拉式菜单，包含文件操作 File（新建、打开、保存等操作），编辑菜单 Edit（复制、粘贴、剪切和删除），编译菜单 Build 中包含时序检查、运行仿真、清理、生成配置文件、多配置文件连接等功能，查看菜单 View 只有放大和缩小两个功能，窗口菜单（Window）可以提供 IEEE-754 的浮点与整型的转换以及打开被关掉的窗口。帮助菜单 Help 包含帮助信息以及注册等。

除了常见的下拉菜单以外，Adaptive IDE 还配备了可视化易操作的一些常见按钮放在工具栏上，可以让用户通过简单点击实现对应操作。每个类别的操作会通过相应的黑色分隔符进行分离。

13.4.2 属性栏

属性栏是实现对在设计空间选中的元素进行配置的操作控件，它既可以配置 Rocell 单元，也可以配置 Busmatrix，根据设计空间中选中的控件不同来展示不同的配置方式。

属性栏对于 Rocell 单元的配置融合了传统的选择操作以及图形化点击操作为一体。每个选中的 Rocell 都可以被配置成指令集中的指令，根据指令的需求，自动选择是否展示 Cin 的配置操作。A，B，Cin 的值都可以通过点击旁边的按键开始选择下方图形中不同的输入方向。也可以再次点击按键直到显示 Constant 或者 Dout。Constant 值可以输入固定值进入配置，Dout 代表着本单元的输入又可以作为输入使用。Rot 代表着 Rotation，也就是配置的时候，这个八角形的单元可以旋转一定的角度，角度每一个整数旋转的度数为 45 度。

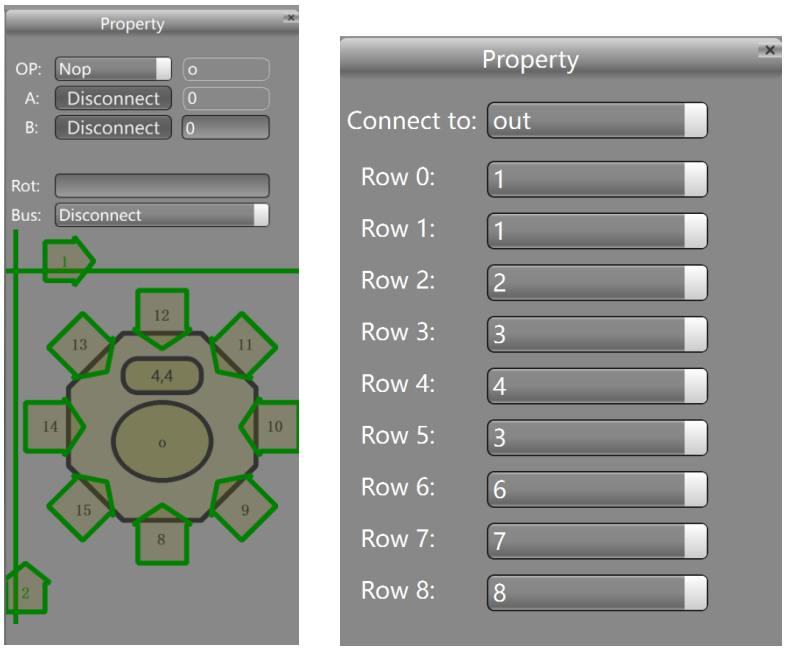


图 13-7 属性栏

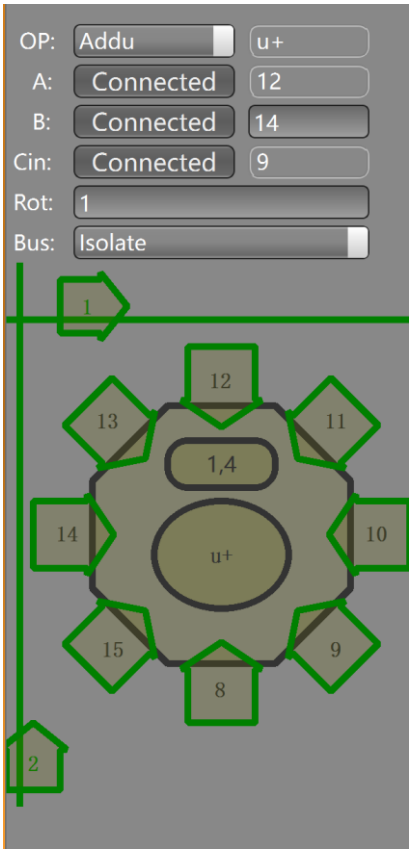


图 13-8 属性栏 Rocell 配置

在选中 Busmatrix 时，属性栏显示出不同于 Rocell 配置的属性，第一行显示的是连接的 Memio 是 Memin 还是 Memout，其余行均为输出端口对应的输入

端口号码。通过选择输出端口对应的输入端，系统会自动在工作空间中进行划线连接。

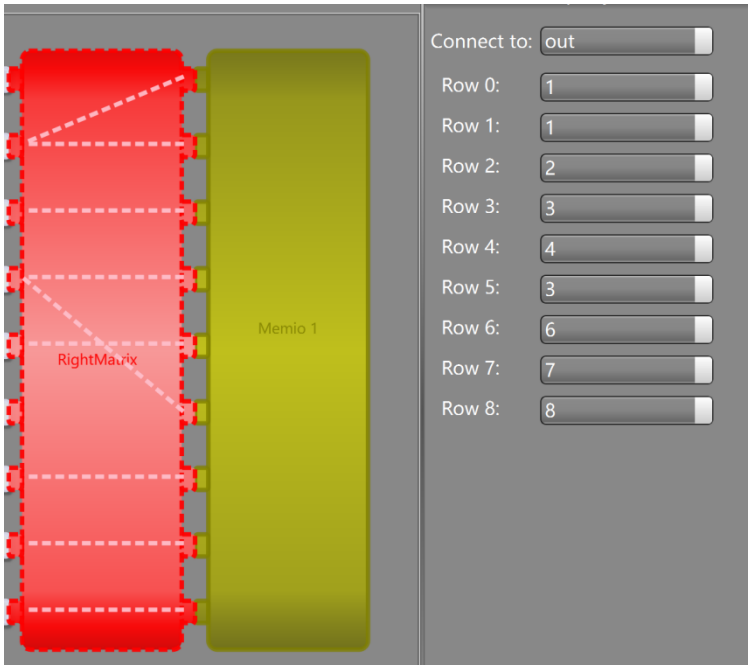


图 13-9 属性栏 Busmatrix 配置

13.4.3 工作空间

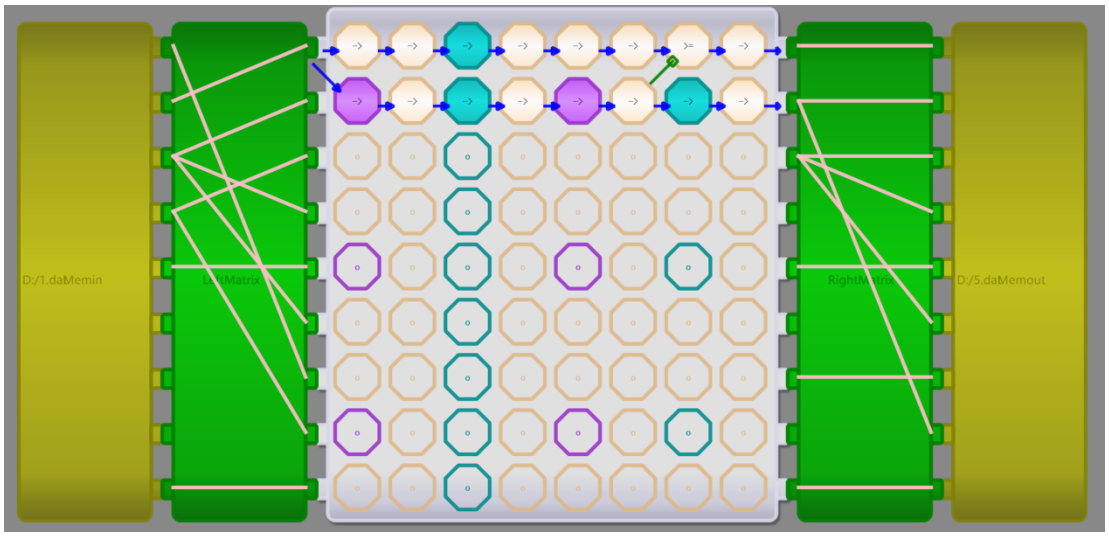


图 13-10 工作空间

工作空间以图形化的形式将每个硬件组件展示出来。针对 Rocell 的配置，每个 Rocell 被选中的时候，属性栏就会将其所有的属性展示出来，用户只需要根据自己的需要将其完成配置即可。无配置的 Rocell 中间将会展示一个小圆圈，外围无填充颜色，代表着没有配置，不启动。每个单元只要操作符不是 Nop，都会参与运算，并消耗对应的能量。当输入 a, b, cin 被配置数据来源的方向时，工作空间会用箭头表示出该单元的数据走向。该 Rocell 所执行的操作指令会用缩写符

号的形式展示出来。比如 Subu 的操作，会用符号“u-”来表示。

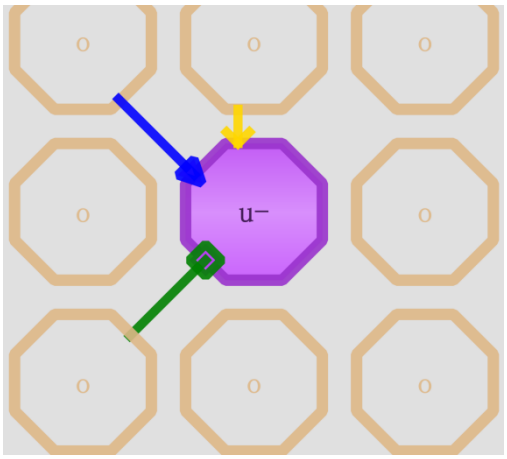


图 13-11 Rocell 配置显示效果

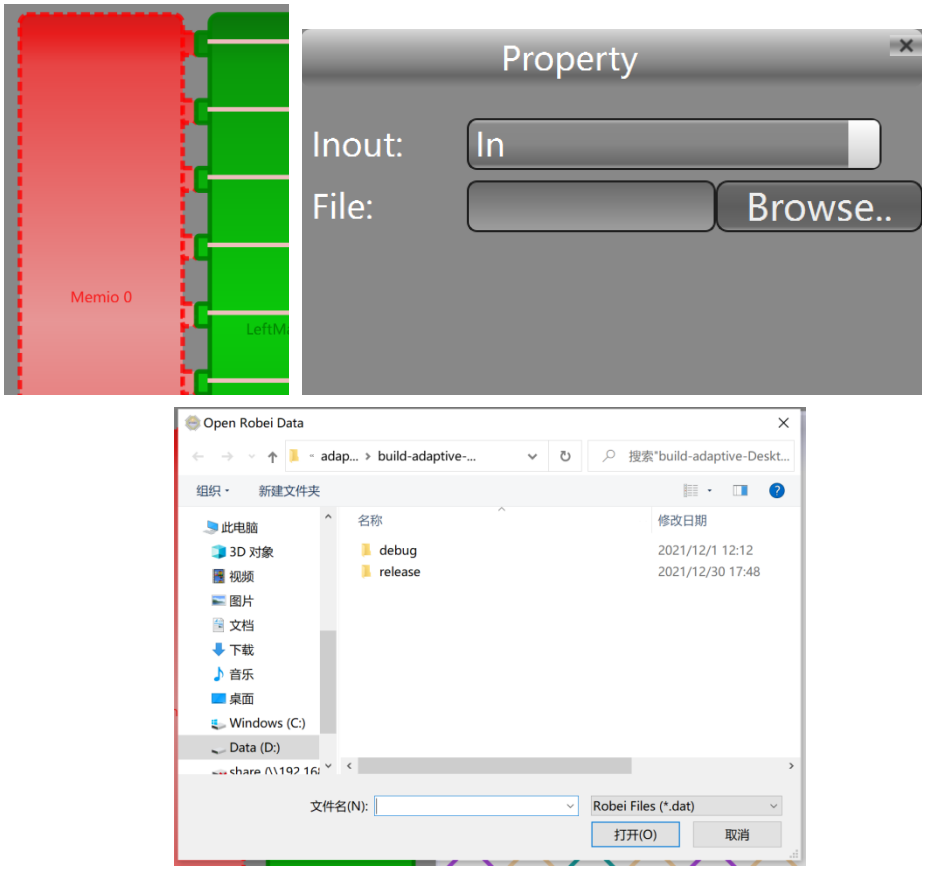


图 13-12 Memio 配置显示效果

选中 Memio，在右侧属性栏配置输入输出方向，然后点击“Browse..”按钮，可以在弹出窗口选择输入或者输出的*.dat 文件，该文件格式模拟 Memio 的存储方式，按照行来进行读写。同一行数据相邻两个数采用“,”进行分割。

1, 2, 3, 4, 5, 6, 7, 8, 9, 10
2, 3, 4, 5, 6, 7, 8, 9, 10, 11
3, 4, 5, 6, 7, 8, 9, 10, 11, 12

图 13-13 *.dat 文件内部数据格式参考示例

13.4.4 特殊操作

设计完成以后，通过点击菜单栏 Build 下拉菜单中的 Check 按钮，可以对整个设计进行延迟路径分析，每个被配置的 Rocell 就会显示从输出端口到当前单元的时序延迟数据。如果出现时序延时不匹配，将会主动在输出窗口进行提示。

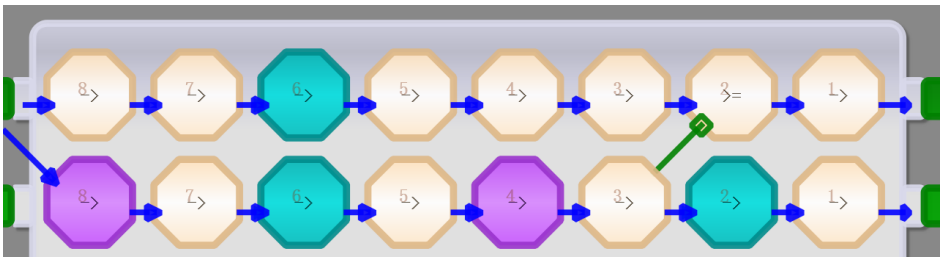


图 13-14 时序延时检查

设计完成以后，通过点击菜单栏 Build 下拉菜单中的 Check 按钮，可以对整个设计进行延迟路径分析，每个被配置的 Rocell 就会显示从输出端口到当前单元的时序延迟数据。如果出现时序延时不匹配，将会主动在输出窗口进行提示。

图 13-15 单步执行

通过菜单栏可以配置单步执行的步长，如果步长不设置，则每次点击运行按钮就会触发整体仿真，一次运行完所有数据。如果设为 1，则按照每次点击运行按钮触发一个时钟进行模拟数据的传输过程。单步执行时，每个单元的上方会按照（a,b）的格式显示输入的数据，底部是输出的数据 X。如果输入 b 的值用常数值，则会显示一个圆圈内部一个小写的 b 代表着输入数据 b 来自于配置的常数。同样，如果 Cin 的值采用的是配置常数，则 Cin 会以一个圆圈加上 c 展示出来。

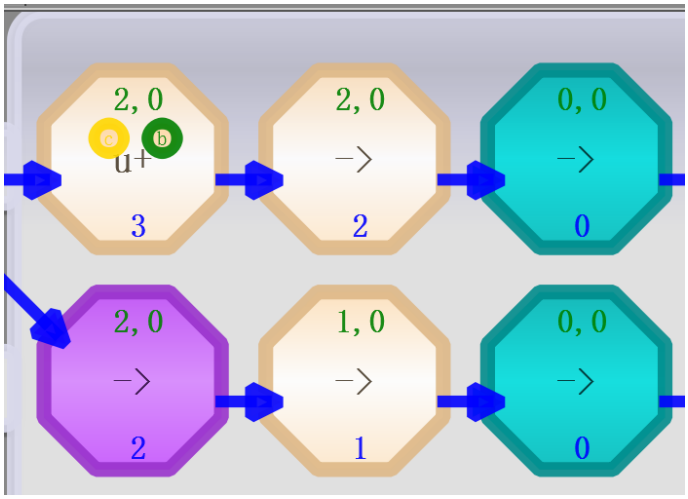


图 13-16 单步仿真

在 Window 菜单栏的下拉菜单中，用户可以选择 Convert，会弹出常用 IEEE-754 标准的浮点和整型互相转换的工具，用户可以输入整型数据或者浮点数进行对应的转换。

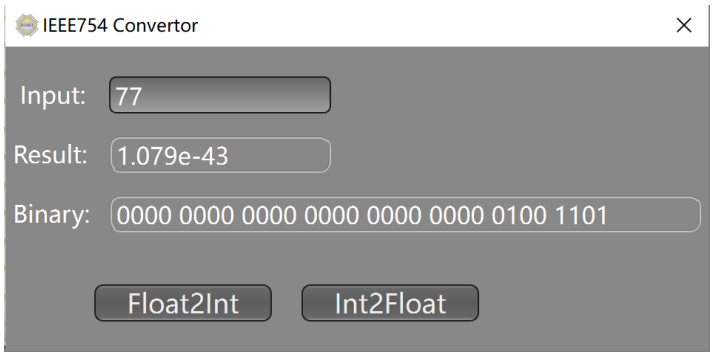


图 13-17 单步仿真

点击 Build 下拉菜单中的 Concat，会弹出 “Concat Bin” 窗口，该窗口允许用户将多个配置的 bin 文件进行连接，按照每个 Bin 文件 1KB 的大小进行连接，然后整合成一个大的 Bin 文件。Input File 作为输入的 bin 文件，可以支持最多 16 个配置文件，每个配置文件的编号作为该文件的索引可以在 Robei IDE 中进行调用，调用函数为：

void Ro_Adaptive_LoadConfig(uint32_t index)；//整体配置 Adaptive

如用户将配置文件放在 Input File 3 中，则在调用该配置的时候，需要调用函数：

Ro_Adaptive_LoadConfig(3)；

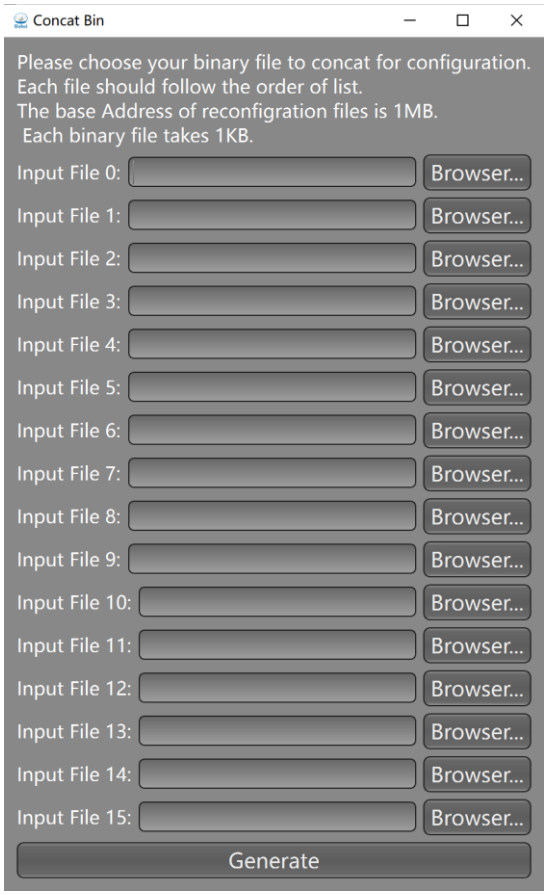


图 13-18 Concat Bin 窗口

14. 符号运算

运用符号进行运算是数学中常见的运算方式，通过公式可以获得输出是输入数据的对应函数运算，一般运算法则可以概括为：

$$Y = f(X) \tag{14-1}$$

其中，X作为输入数据，可以是一个数据，也可以是一组数据。Y作为输出，可以是单个值，也可以是一组值。f作为关系函数，实现了从输入到输出的映射。要实现函数运算，主要是实现f在输入和输出之间的关系表达。

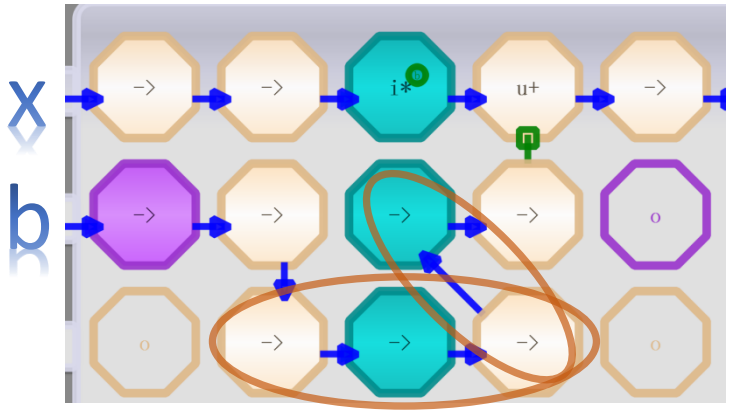
14.1 独立数据

例 14-1 假如独立数据计算公式为

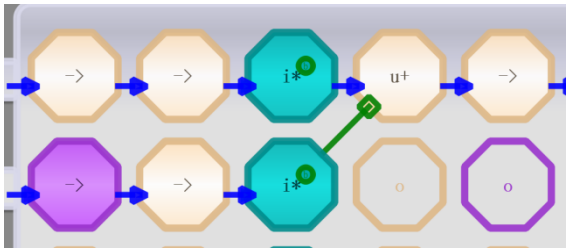
$$y = kx + b \tag{14-2}$$

对于常见的一次函数，假如输入是一个整型数据，输出也是一个整型数据，

中间的k,b也是整数，所采用的四则运算为先乘法后加法，用到指令为 Muli 和 Addu。在 Adaptive 配置中，由于乘法器需要四个时钟周期才出结果，所以如果按照正常的配置模式，b 路传递需要延迟 4 个时钟周期才能匹配 x 路数据的时序，因此我们在 b 路数据进行了绕路行为，增加了 4 个 Rocell 单元（在图 14-1(a)中圈出来的 4 个 Rocell）。实际上，也可以通过 b 路数据乘以 1，通过匹配的乘法器来实现同样的延时，如图 14-1(b)所示。



(a). 采用多个 Rocell 实现延时匹配



(b). 采用乘以 1 来实现 b 路数据的延时匹配

图 14-1 一次线性函数

对于单个数据的计算可以由 CPU 来完成，因为单个数据配置就需要花费不少的时间，然后数据的进出也要消耗时间。对于批量的数据经历同样的公式运算，CPU 的处理速度就明显降低，此时利用 Adaptive 可以实现并行计算，计算采用单路运算，除去配置和进出时间，几乎是每个时钟出一个结果。

14.2 向量数据

在我们计算中，随着信息量的递增，信息越来越多以超越独立数据的形式存在，大量的数据涌入，这就要求普通计算机都拥有信息处理的能力。一维数组是常见的一种数据存储形式，参照 14.1 的案例，对于输入和输出都是一维数组的也可以提炼出类似的形式：

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} = k \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} + m \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} \tag{14-3}$$

公式中，每一个输出都跟对应的输入元素线性相关，如果把输入和输出均看作一

个向量，则可以提炼出：

$$\vec{Y} = k\vec{X} + m\vec{b} \tag{14-4}$$

对于向量的运算，Adaptive 也可以通过简单的配置完成，采用类似图 14-1 的配置方法，在配置上做简单变更即可。输入$\vec{X}$和$\vec{b}$采用 dat 文本输入，每一行代表一个向量，输出$\vec{Y}$也是以行的形式存储。输出文件也是以行的形式进行存储，只需要指定输出文件名称即可。

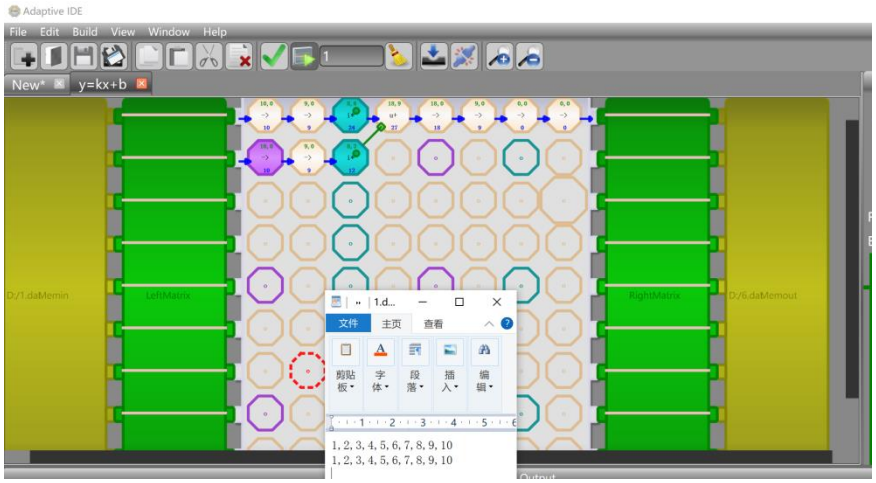


图 14-2 向量线性计算的配置

14.2.1 向量点乘

对于两个向量$\vec{U},\vec{V}$，向量的点乘（内积）代表着两个意义，一是两个向量之间的夹角，另外一个是向量$\vec{V}$在向量$\vec{U}$上的投影。

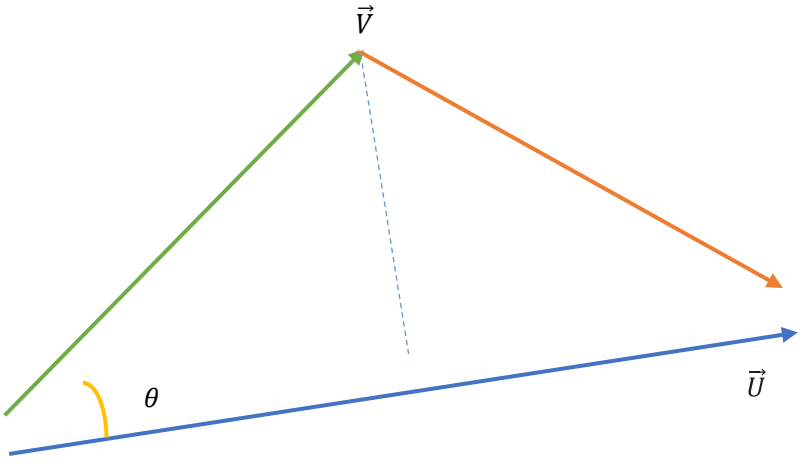


图 14-3 向量点乘的示意图

根据向量点乘的意义，可以列出向量点乘结果为：

$$\vec{U}\cdot\vec{V}=\begin{cases}\|\vec{U}\|\|\vec{V}\|\cos\theta, & if\vec{U}\neq 0\ and\ \vec{V}\neq 0\\ 0, & if\vec{U}=0\ or\ \vec{V}=0\end{cases} \tag{14-5}$$

$\|\vec{U}\|$和$\|\vec{V}\|$表示的是向量的长度， $\cos\theta$ 的正负决定了$\vec{U}\cdot\vec{V}$的正负，因此可以推导出以下结论：

$$\begin{cases}\vec{U}\cdot\vec{V}>0, 向量夹角\ 0<\theta<\frac{\pi}{2}\\ \vec{U}\cdot\vec{V}=0, 向量正交，相互垂直\\ \vec{U}\cdot\vec{V}<0, 向量夹角\frac{\pi}{2}<\theta<\pi\end{cases} \tag{14-6}$$

对于两个任意非零向量，如果计算结果为 0，则可以判断这两个向量是否垂直正交，如果计算结果为正，则代表两个向量大体方向相同，如果计算结果为负，则代表两个向量大体方向相反。这在 GPU 计算中应用比较多。

对于任意维度的向量，我们也可以通过对应数据相乘再相加的方式求出点乘的结果：

$$\vec{U}\cdot\vec{V}=\begin{bmatrix}u_0\\u_1\\\vdots\\u_n\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\\vdots\\v_n\end{bmatrix}=u_0v_0+u_1v_1+\cdots+u_nv_n=\sum_{i=0}^{n}u_iv_i \tag{14-7}$$

具体的配置可以参照图 14-4 所示，其中第一行第四列的 Rocell，可以记为 R(0,3)，需要配置成 Adds 指令，而且让输出结果接到 A 路数据的输入，B 路数据接乘法器输出的值，这样就可以实现一个累加器的效果，但是会将每一步的累加计算结果存储在 Memout 里面，可以到指定的位置去取最终计算结果。如果想要只输出一个最终结果，需要增加配置一个 Mux 指令，在最后一个数据输出的时候选通即可。

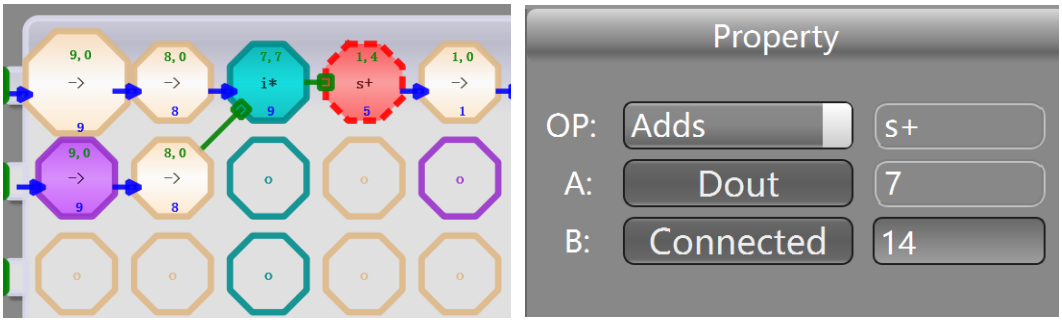


图 14-4 向量点乘的配置

14.2.2 向量叉乘

向量的叉乘求得的是垂直于两个向量所在平面的法向量，也称作两个向量的

外积。外积还有另外一个几何意义就是外积的模在数值上等于由两个向量构成的平行四边形的面积。

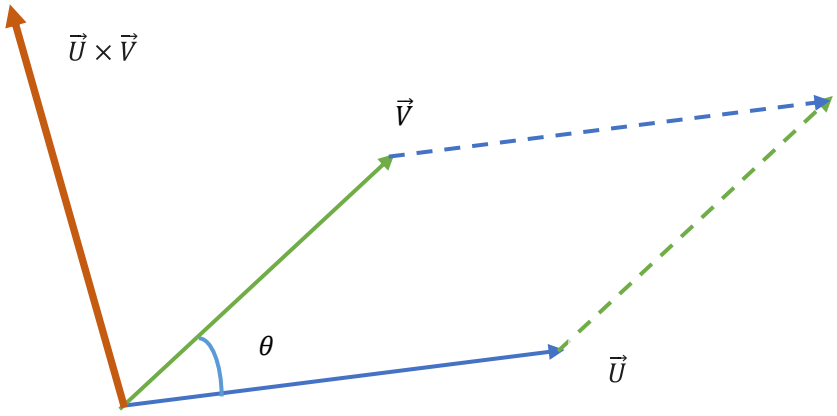


图 14-5 向量叉乘的示意图

以三维向量为例，两个向量的叉乘运算可以简化为：

$$\vec{U}\times\vec{V}=\begin{vmatrix} i & j & k \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix}=(y_1z_2-y_2z_1)i-(x_1z_2-x_2z_1)j+(x_1y_2-x_2y_1)k$$
$$=(y_1z_2-y_2z_1)i+(x_2z_1-x_1z_2)j+(x_1y_2-x_2y_1)k \quad (14\text{-}8)$$

对于多个向量的叉乘，我们采用以下方式来准备 Memio 中存储的数据：

$$\begin{bmatrix}\vec{U}_1\vec{U}_2\cdots\vec{U}_n \\ \vec{V}_1\vec{V}_2\quad\ \vec{V}_n\end{bmatrix}=\begin{bmatrix} u1_x & \cdots & un_x \\ u1_y & & un_y \\ u1_z & & un_z \\ v1_x & & vn_x \\ v1_y & \cdots & vn_y \\ v1_z & & vn_z \end{bmatrix} \quad (14\text{-}9)$$

所匹配的 Adaptive 的配置中，采用 Busmatrix 对对应行的数据进行调用，如第一行 Rocell 的数据，R(0,0)调用的是$u1_y$的数据，因此直接连接 Memin 的第一行数据。同样，Memout 的数据也可以通过 Busmatrix 将数据还原成三维坐标向量。

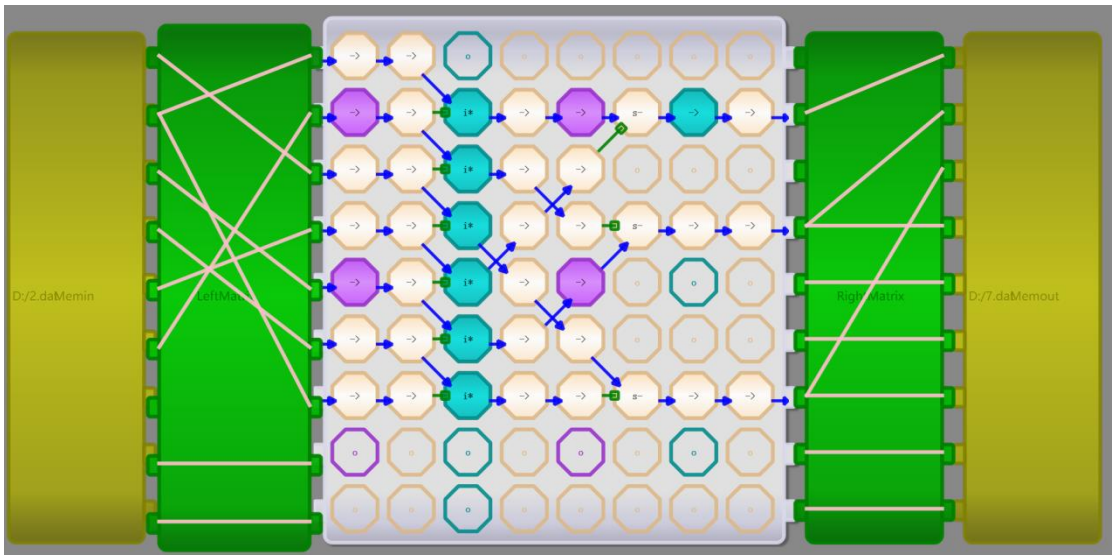


图 14-6 向量叉乘配置图

14.3 软硬件融合下载

如果用户开发的程序不涉及 Adaptive 部分，无须调用 Tool 菜单下的 Adaptive 进行配置。如果用户只需要一种 Adaptive 的配置就可以运行，则无须运行 Concat 进行 bin 文件的连接。针对多个用户所需的 bin 文件进行连接和调用，需要在 Concat 菜单进行操作。这些 bin 文件可以挨着摆放，也可以间隔摆放。如果顺序摆放，生成的文件相对较小，调用的时候只需要按照索引调用即可；间隔摆放，中间没有的文件空间会填充 FF，来满足文件调用的时候只调用 index 即可，这样文件会比较大。如调用图 14-7 中的 Input File 4，vectordot.bin， 在 Robei IDE 中需要调用函数：

Ro_Adaptive_LoadConfig(4)，

而不是调用

Ro_Adaptive_LoadConfig(2)，

因为 2 和 3 的位置被 FF 填充了。

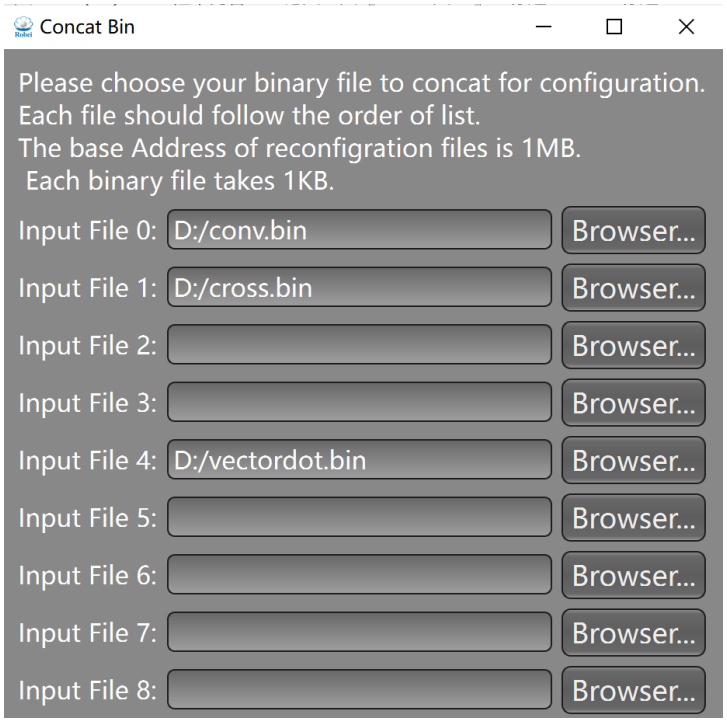


图 14-7 在 Concat 窗口中摆放 bin 文件

14.3.1 Adaptive 全局重构

Adaptive 全局重构是指 Adaptive 的整个配置数据更新，也就是每次全局重构需要从预先写入的 Bin 文件读取配置，加载到 Adaptive 中，实现一次重构。在第一次启动自适应处理器的时候，需要将 Adpative 的中断打开，使能 Adaptive。然后通过 Ro_Adaptive_LoadConfig 函数调用不同的配置进行重构。

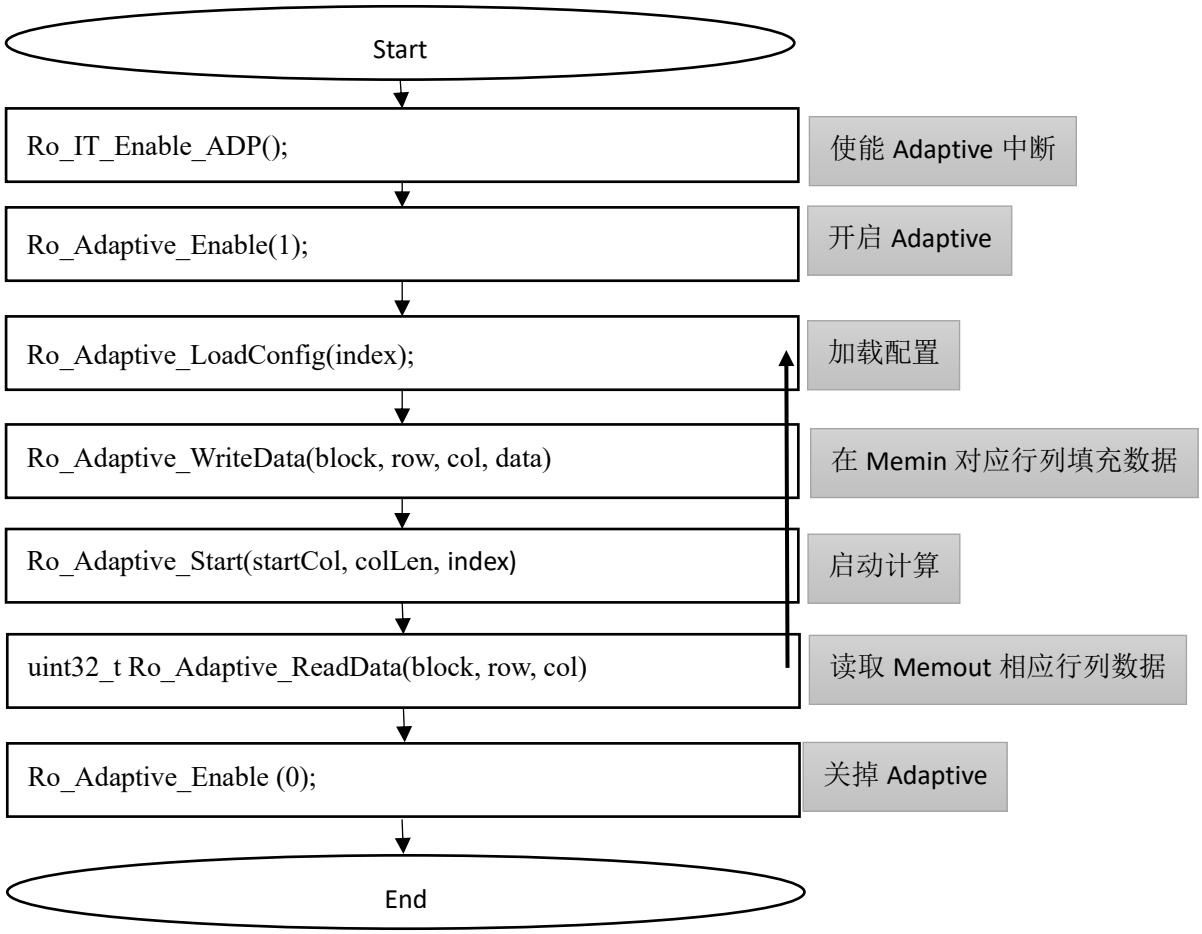


图 14-8 Adaptive 流程图

示例代码：

```
void Ro_MEMIO_KmulXaddB(uint32_t Memindex)
{
    uint8_t u8Direction=MEM_LEFT_TO_RIGHT;
    g_u8AdpIRQFlag=0;
    Ro_IT_Enable_ADP();
    Ro_Adaptive_Enable(1);
    if(!Ro_Adaptive_LoadConfig(Memindex))
        return;
     Ro_Adaptive_MemIOConfig(0,u8Direction);
     Ro_Adaptive_WriteData(0,0,0,1);
     Ro_Adaptive_WriteData(0,1,0,2);
     Ro_Adaptive_WriteData(0,0,1,3);
     Ro_Adaptive_WriteData(0,1,1,4);
     Ro_Adaptive_WriteData(0,0,2,5);
    Ro_Adaptive_WriteData(0,1,2,6);
```

```
    Ro_Adaptive_Start(0,3,Memindex,u8Direction);

    while(1)
    {
        if(g_u8AdpIRQFlag==1)   //中断标志位
        {
            g_u8AdpIRQFlag=0;
            memio_dstbuf.u32Buf[0]= Ro_Adaptive_ReadData(1,0,0);
            memio_dstbuf.u32Buf[1]= Ro_Adaptive_ReadData(1,1,0);
            memio_dstbuf.u32Buf[2]= Ro_Adaptive_ReadData(1,0,1);
            memio_dstbuf.u32Buf[3]= Ro_Adaptive_ReadData(1,1,1);
            memio_dstbuf.u32Buf[4]= Ro_Adaptive_ReadData(1,0,2);
            memio_dstbuf.u32Buf[5]= Ro_Adaptive_ReadData(1,1,2);
            memio_dstbuf.u32Buf[6]= Ro_Adaptive_ReadData(1,0,3);
            memio_dstbuf.u32Buf[7]= Ro_Adaptive_ReadData(1,1,3);
            printf("%08x %08x %08x %08x %08x %08x %08x %08x\r\n",
                    memio_dstbuf.u32Buf[0],
                    memio_dstbuf.u32Buf[1],
                    memio_dstbuf.u32Buf[2],
                    memio_dstbuf.u32Buf[3],
                    memio_dstbuf.u32Buf[4],
                    memio_dstbuf.u32Buf[5],
                    memio_dstbuf.u32Buf[6],
                    memio_dstbuf.u32Buf[7]);
               break;
            }
        }
        Ro_Adaptive_Enable (0);
    }
```

函数注释：

1. 加载配置：Ro_Adaptive_LoadConfig(uint32_t index)

参数 index：代表第几个 bin 文件配置，对应于 Concat 对话框中 Input File 的编号。

2. 写 data 到 Memio：

Ro_Adaptive_WriteData(uint32_t block, uint32_t row, uint32_t col, uint32_t data)

参数 block 是 Memio 的存储块的编号，左侧（memleft）存储块是 0，右侧（memright）存储块是 1。参数 row 是指存储块中第几行，col 是指存储块中第几列。参数 data 是需要写入的数据。

3. 从 Memio 中读 data：

uint32_t Ro_Adaptive_ReadData(uint32_t block, uint32_t row, uint32_t col)

参数 block 是 Memio 的存储块的编号，左侧（memleft）存储块是 0，右侧（memright）存储块是 1。参数 row 是指存储块中第几行，col 是指存储块中第几列。返回读取的坐标位上的数据。

4. 运行 Adaptive：

Ro_Adaptive_Start(uint32_t startCol, uint32_t colLen, uint32_t index, uint32_t direction)

参数：startCol 从第几列开始运算，colLen 是运行多少列，index 是第几个配置，direction 是从左往右还是从右往左进入 Adaptive 计算阵列。

需要处理的数据可以写入 Memio 里面，Memio 有两个块，编号 0 的块是左侧的存储器阵列，编号为 1 的 Memio 是右侧的存储器阵列，每块存储器阵列内部都是以二维存储阵列的形式存在，通过行和列进行定位，按照(block,row,col)的形式进行定位。如（0，0，2）代表着 Memio 0，也就是左侧的 Memio 块，第一行，第 2 列的存储器。

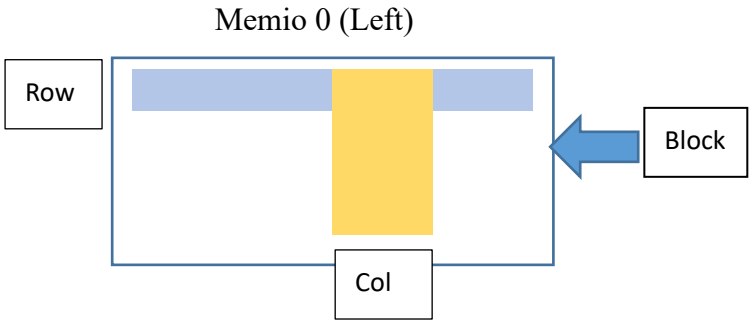


图 14-9 Memio 结构图

14.3.2 Adaptive 局部重构

全局重构所消耗的时间比较长，如果在计算中只需要改变几个单元或者几个配置数据，其余配置不变的话，不需要再重新调用一次新的配置，可以通过局部重构的方式快速实现重构。由于局部重构的配置信息是随着指令加载到 SRAM 中，而且配置数据少，所以操作极快，可以在 10 纳秒~1 微秒级别完成一次局部重构。需要完成局部重构的算法如滤波器的参数，卷积核的权重，线性变换的系数等，可能主体结构变化不大，只需要调整局部结构就可以完成一次计算。

针对 Adaptive 的局部重构可以调用函数：Ro_Adaptive_ConfigCell(row,col,data)来实现单个 Rocell 的重构。行和列代表着 Rocell 的坐标，data 为配置数据。将鼠标放在任一一个 Rocell 上方，就是显示提示信息，对这个 Rocell 的行和列坐标进行标识，同时在右侧属性栏的底部，会显示该 Rocell 的配置函数，直接调用该配置函数即可完成配置。

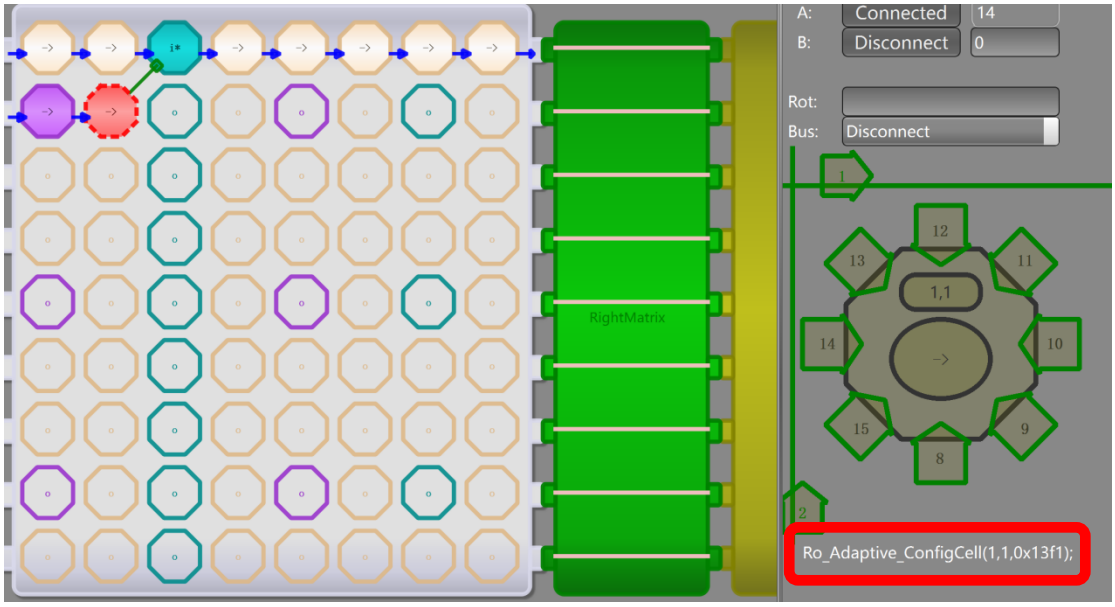


图 14-10 Rocell 配置函数

如果该单元设置了 B 为常数值，则系统会自动再生成一句配置 Breg 为常数值的函数。直接在 Robei IDE 程序中调用该函数就可以完成一个 Rocell 的重构。

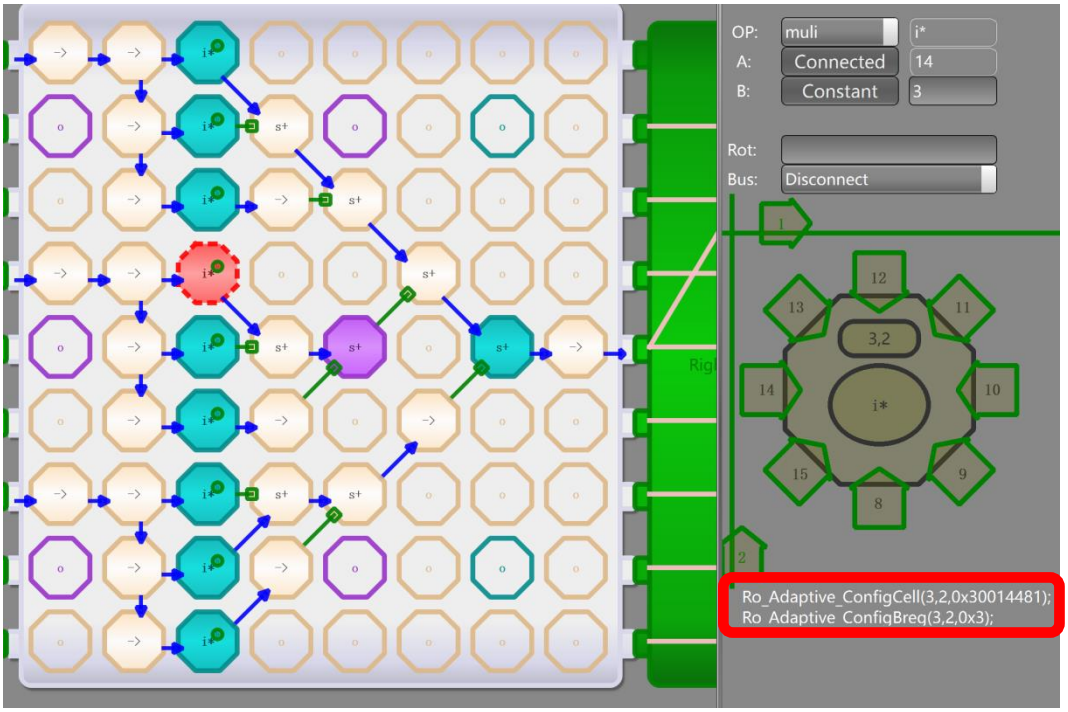


图 14-10 Rocell 配置 Breg

针对 Busmatrix 的配置，在每次点击 Generate 按钮的时候，就会在输出窗口自动输出对应每一行的配置函数，如 Ro_Adaptive_ConfigBusmatrix(0,0x200000); 这个函数实现了对于第 0 行左右两侧的 BusMatrix 的配置。如果哪一行发生了变动，可以只调用对应行的变动的函数即可。

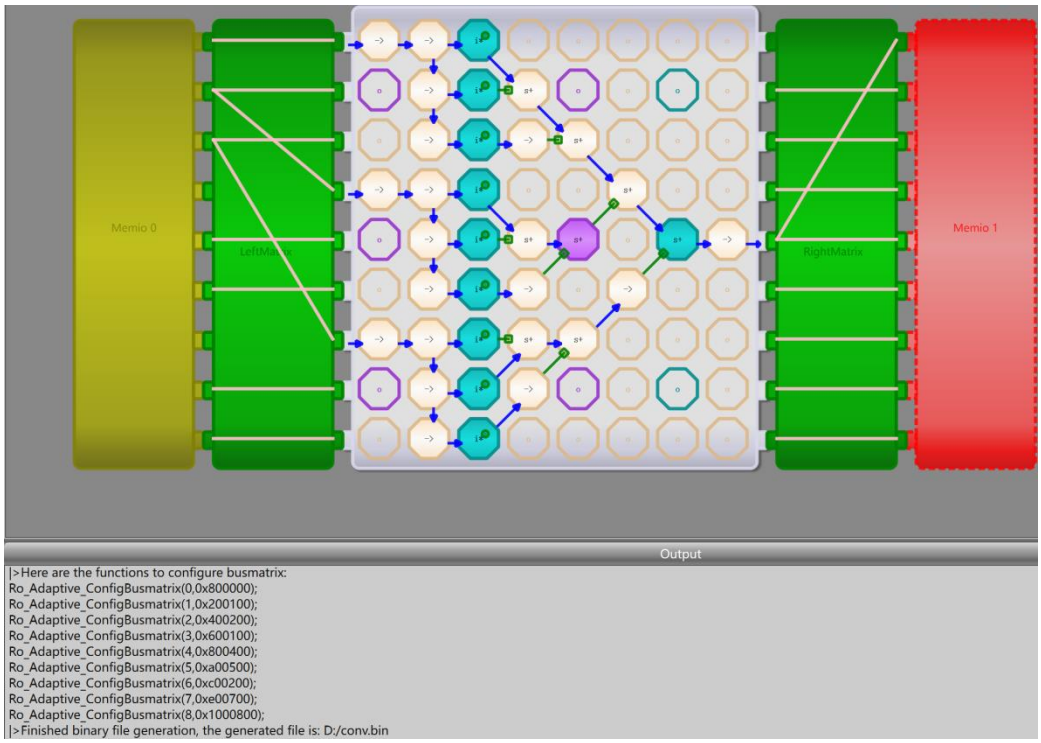


图 14-11 Busmatrix 配置

15. 数字信号处理

数字信号处理及其硬件电路正作为现代电子系统的核心，实现电子系统中复杂的信息处理，并不断替换传统模拟电路。数字系统处理的是数字信号，而真实世界中多为模拟信号，为建立真实世界与电子系统之间的连接，必须通过信号转换。信号转换电路将真实世界的模拟信号转换为数字系统能处理的数字信号，数字信号处理系统则运用各种数字信号处理算法对前端数字信号进行处理，最后将数字系统处理后的数字信号恢复为能为真实世界接受的模拟信号，结构如图 15-1 所示。

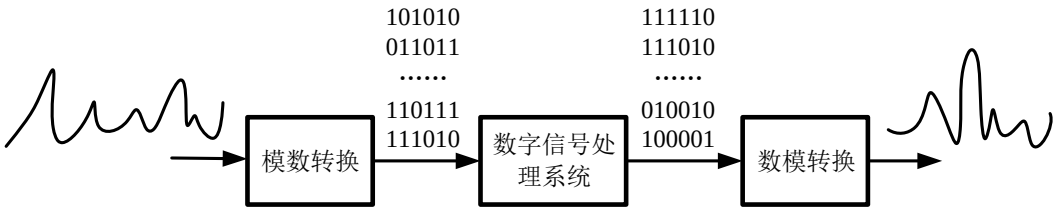


图 15-1 信号处理系统

15.1 滤波器介绍

滤波器最常用于实现消除干扰、获取某个特定频率信号。滤波器有多种分类方法，单从处理信号的类型上来看，可以分为模拟和数字滤波器。模拟滤波器是由电气元件（例如电阻、电容、电感、运放等）组成的电路，对输入的模拟信号进行处理。数字滤波器通过数字信号处理器对输入的数字信号进行运算处理。数字滤波器的输入输出均为数字信号，利用数字运算，例如加、减、乘运算等，改变输入信号从而滤除一些杂波成分。数字滤波器具有精度高、灵活、稳定等优点是模拟滤波器无法媲美的，所以数字滤波器应用的领域更多、更广泛。

15.2 FIR 滤波器

FIR 数字滤波器，指单位脉冲响应长度有限的滤波器，FIR 数字滤波器的单位取样响应 $h(n)$ 是一个 n 点长的有限长序列，$0 \le n \le N-1$。输出 $y(n)$ 可以表示

成输入序列 $x(n)$ 与 $h(n)$ 的线性积。

$$y(n)=\sum_{k=0}^{N-1}x(k)h(n-k)=x(n)*h(n) \qquad (15\text{-}1)$$

系统函数为

$$H(z)=\sum_{n=0}^{N-1}h(n)z^{-n}=h(0)+h(1)z^{-1}+...+h(N-1)z^{-(N-1)} \qquad (15\text{-}2)$$

FIR 数字滤波器是由延迟线加法器和乘法器的集合构成，每一个乘法器的操作系数就是一个 FIR 系数。因为 FIR 数字滤波器输出 $y(n)$ 可以表示成输入序列与范围取样响应的线性卷积，差分方程即系统函数可表示为

$$\begin{aligned} y(n)&=\sum_{k=0}^{N-1}h(k)x(n-k)\\ H(z)&=\sum_{n=0}^{N-1}h(n)z^{-n} \end{aligned} \qquad (15\text{-}3)$$

可以得出 FIR 数字滤波器的直接型结构，如图 14-2 所示

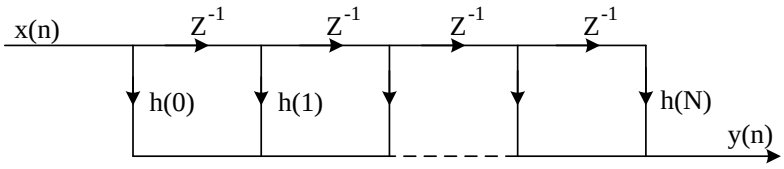


图 15-2 FIR 数字滤波器的直接型结构

本例采用直接型 FIR 滤波器进行演示，RAC102 系列芯片可以实现 8 级 FIR 滤波器，由于每个 Rocell 自带一个延时，数据可以直接像流水瀑布一样从上往下流下，通过对应配置系数的乘法器之后，再累加起来输出。其中，参数配置按照自下而上的模式进行配置，因为第一个数值对应的乘法器在最下方。

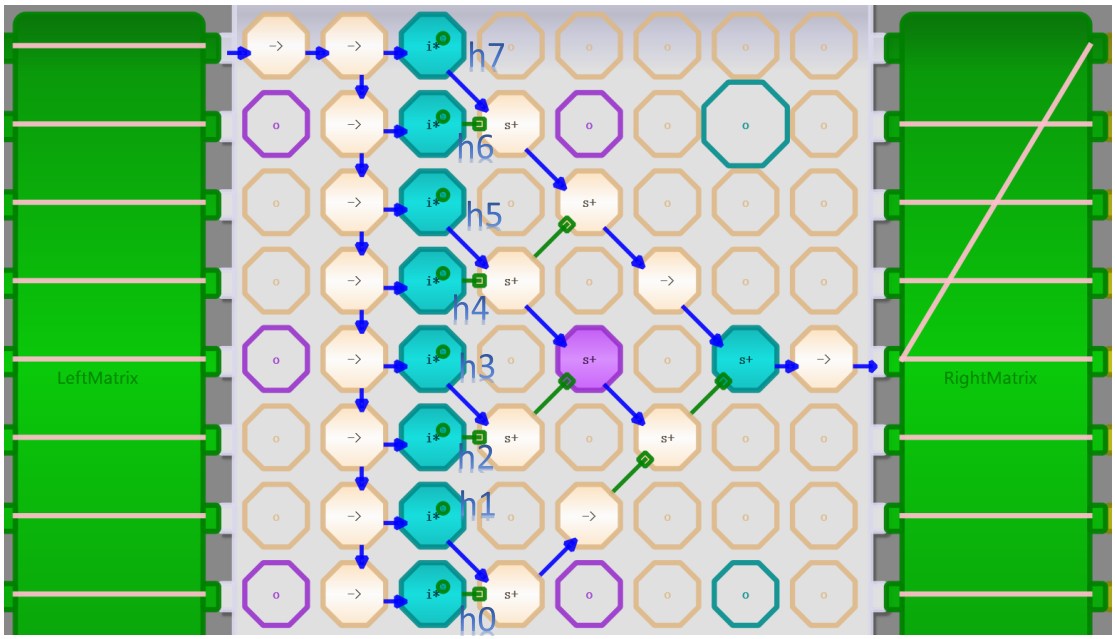


图 15-3 FIR 数字滤波器的配置

16. 卷积运算

16.1 卷积运算的原理

卷积（**Convolution**）是一种数学操作，采用两个同样大小数据阵列进行一对一相乘然后再将所有乘积相加。卷积也经常被看作是一种滤波算法，卷积核(kernel)用于过滤特征映射以找到某种信息，比如一个卷积核可能只找边缘，而抛掉其他信息。卷积计算如式中所示，可以理解为 3 个向量点乘后再相加。由此可以得到卷积运算公式为：

$$
\begin{aligned}
Conv（W,X）&= \begin{bmatrix} w00 & w01 & w02 \\ w10 & w11 & w12 \\ w20 & w21 & w22 \end{bmatrix} * \begin{bmatrix} x00 & x01 & x02 \\ x10 & x11 & x12 \\ x20 & x21 & x22 \end{bmatrix} \\
&= w00 * x00 + w10 * x10 + w20 * x20 + \\
&\quad w01 * x01 + w11 * x11 + w21 * x21 + \\
&\quad w02 * x02 + w12 * x12 + w22 * x22 \\
&= W_0 \cdot X_0 + W_1 \cdot X_1 + W_2 \cdot X_2
\end{aligned} \tag{16-1}
$$

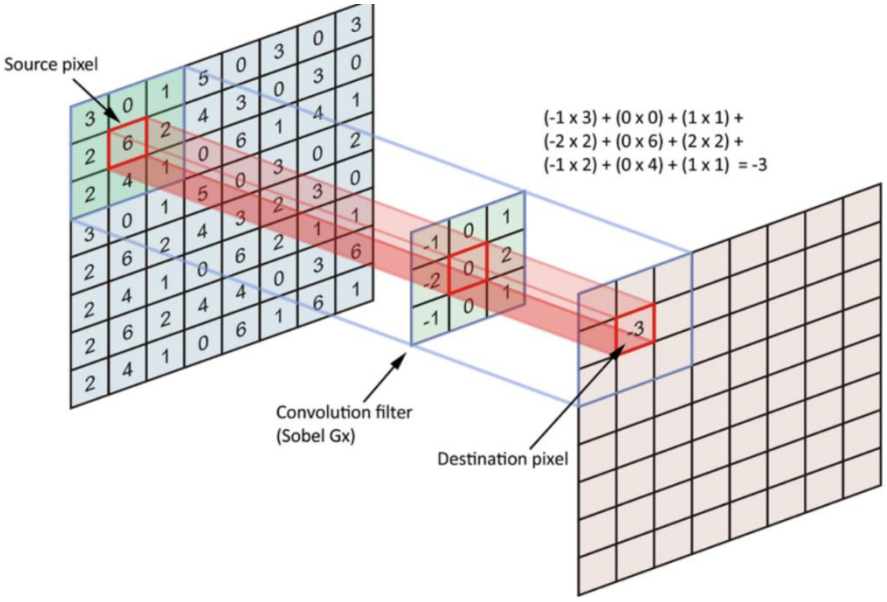


图 16-1 卷积运算

图像处理中，卷积核可以从左到右不断滑动，每次滑动一列，每次都有一列新值加进来做运算，运算结果为一个像素，存储到新图片上。

16.2 卷积核实现

参照上节描述，Memio 中图像数据（m 行 n 列）的存储格式应该为：

$$I = \begin{bmatrix} x_{00} & \cdots & x_{0n} \\ \vdots & \cdots & \vdots \\ x_{m0} & \cdots & x_{mn} \end{bmatrix} \tag{16-2}$$

本例采用 3x3 的卷积核来实现对应的配置设计，因此，卷积运算可以一次进入 Adaptive 三行（一个窗口的高度），每次运行完三行运算后，可以通过动态配置 Busmatrix 来实现三行数据往下偏移一行（下一个窗口），依此类推，输出也是逐步递增存储结果的行数。这样就可以实现最小的动态重构时间完成最快的卷积计算。

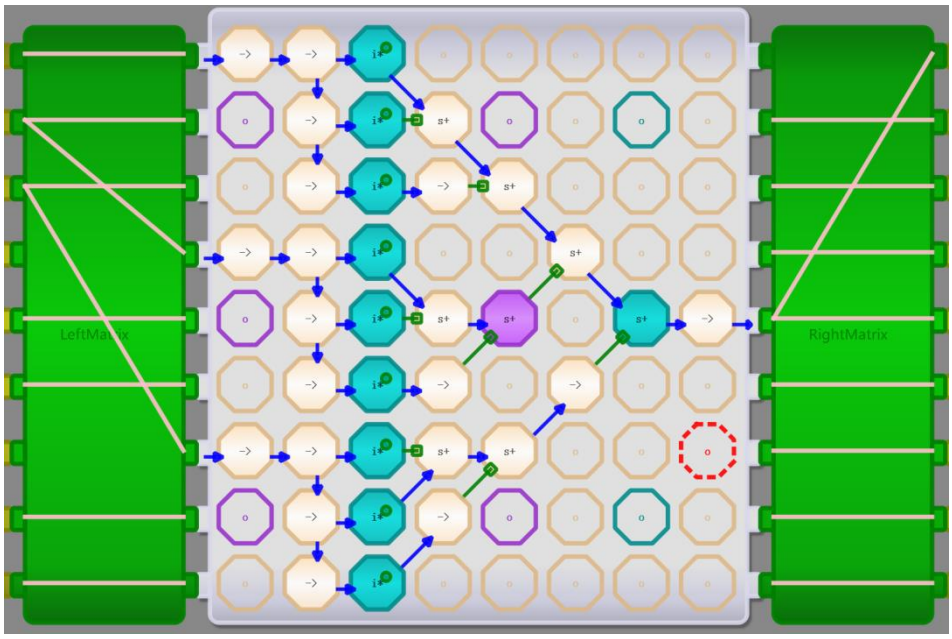


图 16-2 Adaptive 卷积核实现

对于每个窗口数据的卷积计算，可以拆解成每行数据的点乘运算，最后再相加。由于每个 Rocell 自带一个延时，因此可以通过数据向下传递的方式实现三级延时。通过同时乘以相应的权重，最后相加可以得到一个窗口的卷积值。随着数据的流水式涌入，可以实现卷积核的窗口式滑动。

每计算完三行的输入，可以重构一下 Busmatrix，让左侧 Busmatrix 连接 Memio 0 的三个输入每个都下移一行，就可以实现窗口换行运行，此时右侧 Busmatrix 的 Memio1 也需要将输出下移一行。如果全部 Memio 的数据都计算完了，就可以将整个 Memio 输出，再截取另外一段数据进行计算。

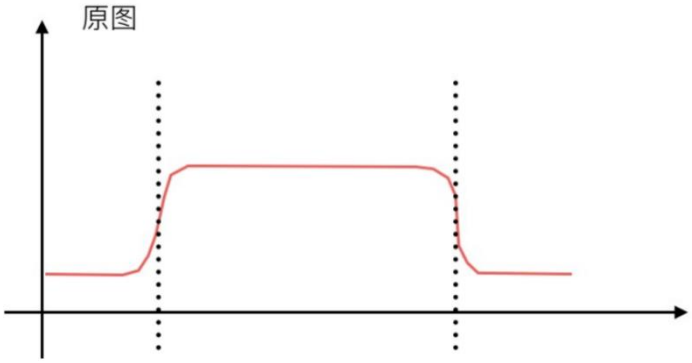
卷积核的重构主要是权重的变化，其计算架构不会发生变化。因此，每次的变动只需要更新 9 个 Rocell 的 breg 值即可，也就是说每次重构只需要 18 个时钟周期即可完成重构。根据 3x3 卷积核的权重不同，我们可以得到不同的重构算法。

16.3 图像滤波

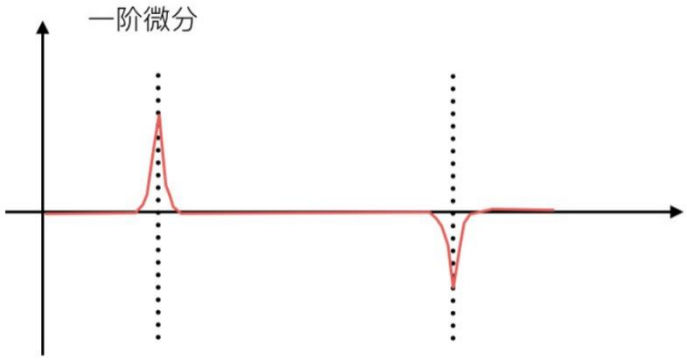
图像滤波器其实就是借助像素点自己和临近位置数据进行卷积运算来修正像素信息。在一幅图像中，临近像素点不仅是上下左右像素点，还包括斜角上像素点、甚至有些还要包含隔一到几个位置像素点的信息。图像滤波器通常采用卷积来实现，滤波器也有不同尺寸，比如 3x3、5x5 等。

16.3.1 边缘检测

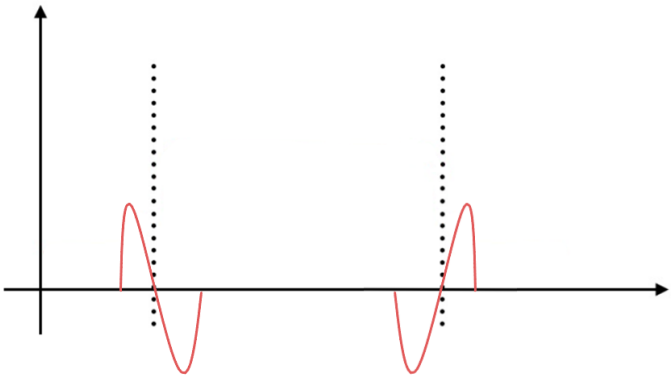
图像最基本特征之一就是图像边缘，也就是周围像素灰度有跳跃性变化的像素集合。边缘是图像局部强度变化最明显的地方，因此它是图像分割中一种重要特征。图像边缘有方向和幅度两个属性，比如沿着边缘方向上，像素值变化比较平缓，但是垂直于边缘方向上，像素变化非常剧烈。这种梯度变化特征可以通过微分算子检测出来，一般用一阶或两阶导数来检测边缘特征。一阶导数认为最大值对应于边缘位置，而二阶导数则通过过零点来对应边缘位置。



（a）带有边缘的原图，在一定位置发生了突变[13]



（b）经过一阶微分之后，发生突变地方会产生明显与周围位置不同的数值[13]



（c）经过二阶微分之后，边缘位置是明显的过零点

图 16-3 微分可以提取出边缘特征

一幅图像是二维离散矩阵，因此，对图像进行求梯度变换等于对每个方向上分别求导，如：

$$|\text{grad}(x,y)| = \sqrt{(\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial x})^2} \tag{16-3}$$

在计算机中由于乘法和开放都比较耗费时间，计算复杂度很高，为了降低计算量，可以用近似绝对值来表达近似的梯度幅值：

$$|\text{grad}(x,y)| \approx \left|\frac{\partial f}{\partial x}\right| + \left|\frac{\partial f}{\partial y}\right| \tag{16-4}$$

因此，只要分别计算出在横向和纵向一阶导数，就可以找到梯度变化近似值。一阶导数在离散数据中也叫做差分，公式为：

$$\frac{\partial f}{\partial x} = f(x+1,y) - f(x,y) \quad , \quad \frac{\partial f}{\partial y} = f(x,y+1) - f(x,y) \tag{16-5}$$

对于一个 3x3 模板如式(5) 所示，

$$I = \begin{bmatrix} f(x-1,y-1) & f(x,y-1) & f(x+1,y-1) \\ f(x-1,y) & f(x,y) & f(x+1,y) \\ f(x-1,y+1) & f(x,y+1) & f(x+1,y+1) \end{bmatrix} \tag{16-6}$$

可以通过类似方式计算出分别在 x 方向和 y 方向的偏导数：

$$\begin{aligned} \frac{\partial f}{\partial x} &= f(x,y-1) - f(x-1,y-1) + f(x,y) - f(x-1,y) + f(x,y+1) - f(x-1,y+1) + \\ &\quad f(x+1,y-1) - f(x,y-1) + f(x+1,y) - f(x,y) + f(x+1,y+1) - f(x,y+1) \\ &= f(x+1,y-1) - f(x-1,y-1) + f(x+1,y) - f(x-1,y) + f(x+1,y+1) - f(x-1,y \\ &\quad +1) \\ &= \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} f(x-1,y-1) & f(x,y-1) & f(x+1,y-1) \\ f(x-1,y) & f(x,y) & f(x+1,y) \\ f(x-1,y+1) & f(x,y+1) & f(x+1,y+1) \end{bmatrix} \end{aligned} \tag{16-7}$$

$$\begin{aligned} \frac{\partial f}{\partial y} &= f(x-1,y) - f(x-1,y-1) + f(x,y) - f(x,y-1) + f(x+1,y) - f(x+1,y-1) + \\ &\quad f(x-1,y+1) - f(x-1,y) + f(x,y+1) - f(x,y) + f(x+1,y+1) - f(x+1,y) \\ &= f(x-1,y+1) - f(x-1,y-1) + f(x,y+1) - f(x,y-1) + f(x+1,y+1) - f(x \\ &\quad +1,y-1) \end{aligned}$$

$$= \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} f(x-1,y-1) & f(x,y-1) & f(x+1,y-1) \\ f(x-1,y) & f(x,y) & f(x+1,y) \\ f(x-1,y+1) & f(x,y+1) & f(x+1,y+1) \end{bmatrix} \quad (16\text{-}8)$$

因此，水平方向和垂直方向梯度卷积算子可以提取出来，这两个算子也称为 Prewitt 算子：

$$P_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, P_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad (16\text{-}9)$$

应用 Prewitt 算子计算边缘检测结果如图 16-4 所示，输入权重矩阵P_x为：

```
w00=-1; w01=0; w02=1;
w10=-1; w11=0; w12=1;
w20=-1; w21=0; w22=1;
```

输入的权重矩阵P_y为：

```
w00=-1; w01=-1; w02=-1;
w10=0; w11=0; w12=0;
w20=1; w21=1; w22=1;
```

(a) P_x运行结果　　(b) P_y运行结果

图 16-4　Prewitt 算子运算结果

Sobel 边缘检测是由 Irwin Sobel 提出来在梯度卷积算子之上加入高斯变换卷积算子，也就是采用了对于中间一行和中间一列算子放大两倍来实现。Sobel 算子也有水平和纵向两种不同算子，但是数值差不多，只不过是做了 90 度旋转，如式（16-10）所示。

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad (16\text{-}10)$$

因此，修改图 16-4 中卷积模型 Parameters 即可实现 Sobel 边缘检测。要实现垂直方向边缘检测与提升，其权重矩阵为：

w00=-1; w01=0; w02=1;

w10=-2; w11=0; w12=2;

w20=-1; w21=0; w22=1;

要实现垂直方向 Sobel 边缘检测，权重矩阵可以修改为：

w00=-1; w01=-2; w02=-1;

w10=0; w11=0; w12=0;

w20=1; w21=2; w22=1;

所得到的图像结果如图 16-5 所示。G_x检测出来的是纵向边沿，是沿着水平方向梯度变化；而 G_y实现的是水平方向边缘检测，沿着垂直方向梯度变化。

(a) G_x运行结果　　(b) G_y运行结果

图 16-5　Sobel 算子运算结果

同样的边缘检测还有 Laplacian、Marr 算子，这两种算子都属于二阶微分算子，其中 Laplacian 算子来自于拉普拉斯变换。

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial_x^2} + \frac{\partial^2 f}{\partial_y^2} \tag{16-11}$$

其中，

$$\frac{\partial^2 f}{\partial_x^2} = \big(f(x+1,y) - f(x,y)\big) - \big(f(x,y) - f(x-1,y)\big) = f(x+1,y) - 2f(x,y) + f(x-1,y) \tag{16-12}$$

$$\frac{\partial^2 f}{\partial_y^2} = \big(f(x,y+1) - f(x,y)\big) - \big(f(x,y) - f(x,y-1)\big) = f(x,y+1) - 2f(x,y) + f(x,y-1) \tag{16-13}$$

因此，

$$\begin{aligned}\nabla^2 f(x,y) &= f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y) \\ &= \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} f(x-1,y-1) & f(x,y-1) & f(x+1,y-1) \\ f(x-1,y) & f(x,y) & f(x+1,y) \\ f(x-1,y+1) & f(x,y+1) & f(x+1,y+1) \end{bmatrix}\end{aligned} \tag{16-14}$$

可以通过式（16-14）提炼出拉普拉斯算子：

$$\mathcal{L} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{16-15}$$

二阶算子包含了对横向二阶偏微分和纵向二阶偏微分，因此不需要分成两个通道，可以用一个算子来获得两个方向上边缘检测。图像中一个较暗区域中如果有一个亮点，用拉普拉斯运算就会使这个亮点变得更亮，因为图像中边缘就是灰度发生跳变的区域。因此特别适用于以突出图像中孤立点、孤立线或线端点为目的的场合。拉普拉斯算子也会增强图像中噪声，有时用拉普拉斯算子进行边缘检测时，可将图像先进行平滑处理。

Marr 算子也称为 Log 算子，是先进行高斯滤波再进行拉普拉斯算子检测，然后找过零点来确定边缘位置。Marr 算子比拉普拉斯算子多了斜方向梯度，四个角落均有权重。采用 Robei 分别执行拉普拉斯与 Marr 算子得到图像结果如图 16-6 所示。

$$\mathcal{M} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{16-16}$$

(a) Laplacian 运行结果　　(b) Marr 运行结果

图 16-6 Laplacian、Marr 算子运算结果

16.3.2 图像平滑

图像平滑从信号处理角度看就是过滤掉其中高频信息，保留低频信息。而边缘检测其实是采用高通滤波器，过滤掉低频信息，保留高频信息。图像平滑算法有很多种，本节介绍比较常见的均值滤波。

均值滤波顾名思义，就是采用卷积核覆盖区域像素平均值代替卷积核中心点对应像素的值。如果采用 mxn 卷积核（m、n 为奇数），卷积核中间值为f(x,y)，

中心点距离两边行数以及列数为a = （m − 1）/2，b = （n − 1）/2，则其计算公式为：

$$\text{Average}(x,y) = \frac{1}{m*n}\sum_{t=-a}^{a}\sum_{s=-b}^{b} f(x+t, y+s) \tag{16-17}$$

以 3x3 的卷积核为例，可以得到：

$$\text{Median}(x,y) = \frac{1}{3*3}\sum_{t=-1}^{1}\sum_{s=-1}^{1} f(x+t, y+s)$$

$$= \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} * \begin{bmatrix}f(x-1,y-1) & f(x,y-1) & f(x+1,y-1)\\ f(x-1,y) & f(x,y) & f(x+1,y)\\ f(x-1,y+1) & f(x,y+1) & f(x+1,y+1)\end{bmatrix} \tag{16-18}$$

可以提取出均值滤波卷积算子为

$$\text{Median} = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} \tag{16-19}$$

将均值滤波应用到 Robei 灰度图上，除了将卷积核权重矩阵改为全 1 以外，还要在卷积最后一步，采用移位的方式实现除法运算，如式 16-19 虽然是除以 9，在不精度许可的情况下，可以采用右移 3 来实现除以和 9 接近的值 8。结果为图 16-7 所示图像。下述其他平滑滤波器也类似，将权重和在最后一步中除去。

图 16-7 均值滤波

除了均值滤波以外，比较常用的还有加权均值滤波器，也就是对卷积算子每一位进行加权再与像素卷积，然后将结果除以所有权重的和，具体公式可以表示为：

$$\text{MedianW}(x,y) = \frac{\sum_{t=-a}^{a}\sum_{s=-b}^{b} f(x+t,y+s)}{\sum_{t=-a}^{a} w(s,t)} \tag{16-20}$$

高斯滤波器原理和均值滤波器有些类似，都是采用滤波器窗口内像素均值作为输出。但是高斯滤波窗口模板系数和均值滤波器不同，均值滤波器模

板系数都是 1；而高斯滤波器模板系数，是随着离模板中心的距离增大而减小，类似于高斯分布函数，越远离中心位置数据值越小。

$$h(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{16-21}$$

对于卷积窗口大小为（2k+1）x（2k+1）的卷积核，模板中各个元素值计算方式为：

$$h(i,j) = e^{-\frac{(i-k-1)^2+(j-k-1)^2}{2\sigma^2}} \tag{16-22}$$

式 16-22 计算出来都是小数，但是在基于 Adaptive 的图像卷积中，尽量用整数计算来加快速度，因此，可以对计算结果归一化，使用模板左上角系数的倒数作为归一化系数$z = \frac{1}{w00}$（左上角系数值被归一化为 1），模板中每个系数都乘以z，然后将得到的值取整，就得到了整数型高斯滤波器模板。以 $\sigma = 0.8$ 的 3x3 的卷积算子为例：

$$\text{Gauss} = \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix} \tag{16-23}$$

应用高斯平滑算子到图像上，可以得到如图 16-8 所示结果。

图 16-8 高斯滤波

17. 自适应架构特色

以控制为核心的计算简称为控制流计算，也就是核心调度的是控制指令。以数据处理为核心的计算简称为数据流计算，也就是核心调度的是数据。传统的

CPU 架构采用控制器读取指令、解析指令，然后从存储器调度数据进行计算的模式，这种模式实现的是通过指令来处理数据的控制流计算。在几十年前，硅资源受限的情况下，这种模式通过调度处理的方式节约了硅的面积，降低了成本，顺应当时的市场发展需求。随着数据量的递增，传统的单颗 CPU 架构无法处理海量数据的并行化计算，才引入了多核 CPU 的并行化处理，这种多核处理器的堆积只是将控制流计算实现了倍增，而不是面向大量数据的数据流计算。

随着大数据时代的发展，传统的控制流计算跟不上海量数据处理的需求，数据流计算架构才应时而生。沉芯芯片的自适应处理阵列就是这样一种数据流的计算方式，采用了大规模阵列式处理器，通过重配置实现不同的数据运算和数据流向，让数据流入，流出就是结果。控制流计算适合于串行计算，因为有大量的数据跳转分支等，数据流计算适合于并行计算，同时海量数据流入和流出，流的过程就实现了计算。

图 17-1 传统架构与自适应对比

要想完美的融合控制流与数据流计算，需要将两种处理器融合为一体，方能自由调度与控制。沉芯系列芯片的优势在于将串行处理的 CPU 架构和并行处理的自适应架构融合为一颗芯片，实现串并行一体化，将调度交给 CPU，处理交给自适应，各司其职，追求性能与成本的最优。

图 17-2 沉芯异构芯片架构

通过数据流计算与控制流计算的深度融合，打破传统 CPU 通过指令处理数据的方式，改为指令调度配置，配置处理数据。这些配置其实可以对应看作为相应的指令，对于单个数据要经过 K 次处理（K 条指令），要想实现最低的移动成本（读写能耗最低），此时指令多，数据少，让数据反复进出是能耗最低的办法。当数据变成 M 倍且每个数据都执行相同的 K 次处理，M>>K 的时候，数据远远超过指令的数据，此时最有效的调度就要变为调度指令了。

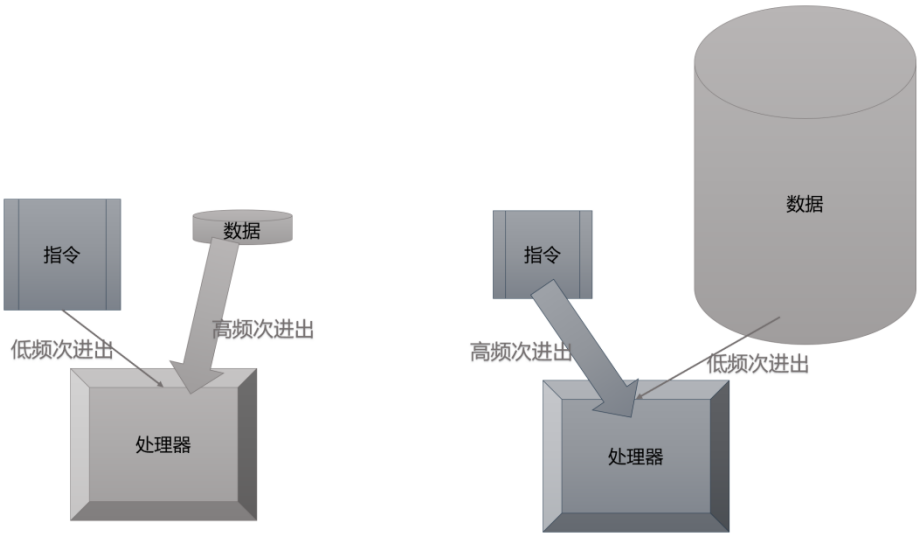


图 17-3 指令与数据的大小不同，决定了调度的不同

对于数据的吞吐来讲，传统的 CPU 架构受限于总线的位宽，海量数据的并行吞吐能力差，采用沉芯异构架构以后，由于配置数据减少，总线位宽实现对自适应配置的调度绰绰有余，数据存储在 Memio 中，可以以巨大的带宽进出自适应并行处理器阵列，而且可以通过乒乓操作，数据不用出来，直接在内部配置完自适应架构后反复进出自适应，最后将结果输出。

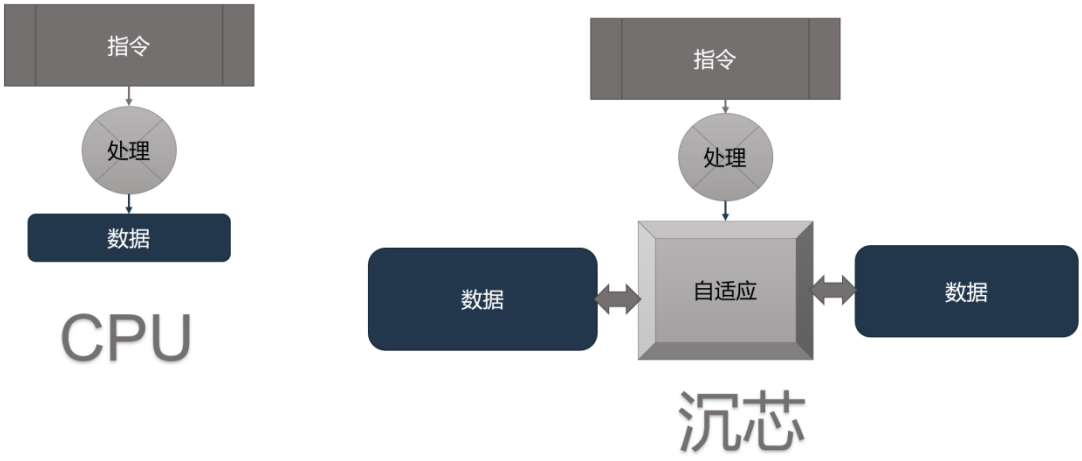


图 17-4 CPU 调度与沉芯调度的不同

沉芯异构架构和传统的 CPU 以及 FPGA 都是基于硅工艺制造，可算是同一物种，传统的 CPU 就如同蜥蜴，一次成型永不改变，有任何的变化这个芯片就废了。FPGA 是一种可以软件编程的蜥蜴，但是配置周期比较长，重构一个 Die 需要几百毫秒的时间，可以认为是一种人工刷漆的方式实现蜥蜴的变色。沉芯自适应芯片可以在微秒到纳秒级别实现一次重构，重构的时间是 FPGA 的万分之一不到，一眨眼就可以重构 10 万次以上，这个级别的重构完全通过软件调度来完成，依据软件的需求调度不同的硬件算法，这是一种变色龙的方式，是真正有别于 CPU 和 FPGA 的新型处理器架构。

类别	厂家	图形描述	优势	缺点
ASIC	Intel，NVidia，Qualcomm，Huawei		1.知名度高 2.制程工艺好 3.市场占有率高 4. 资金雄厚 5. 软件生态完善 6. 串行处理达到极致	1. 不可重构 2. 并行处理差 3. 功耗高 4. 硬件更新难 5. 按需定制难
FPGA	Intel（Altera），Xilinx，Lattice		1. 可静态重构 2. 并行加速 3. 可实现任何逻辑 4. 部分动态重构 5. 市场认可度高 6. 实力雄厚	1. 需要上电擦除 2. 重构速度慢 3. 功耗高 4. 执行频率低 5. 设计复杂 6. 成本高
Adaptive	Robei		1. 高速动态重构 2. 并行加速 3. 硬件可更新 4. 一芯多用 5. 设计简单 6. 功耗低 7. 串并行一体化 8. 成本可控	1. 应用资源少 2. 需要培养市场 3. 需要建立生态 4. 需要不断研发更新

图 17-5 同是一种物种，不同的特性

任何一种处理器都有其特色，但是往往这种特色带来了其他的短板。芯片不是一个理论工程，而是一个系统工程，充斥着大量的折中。一味追求某一个性能的突出，往往带来了其他性能的制约。沉芯自适应架构追求的是折中与平衡，尽量补齐其短板，实现成本最优化。

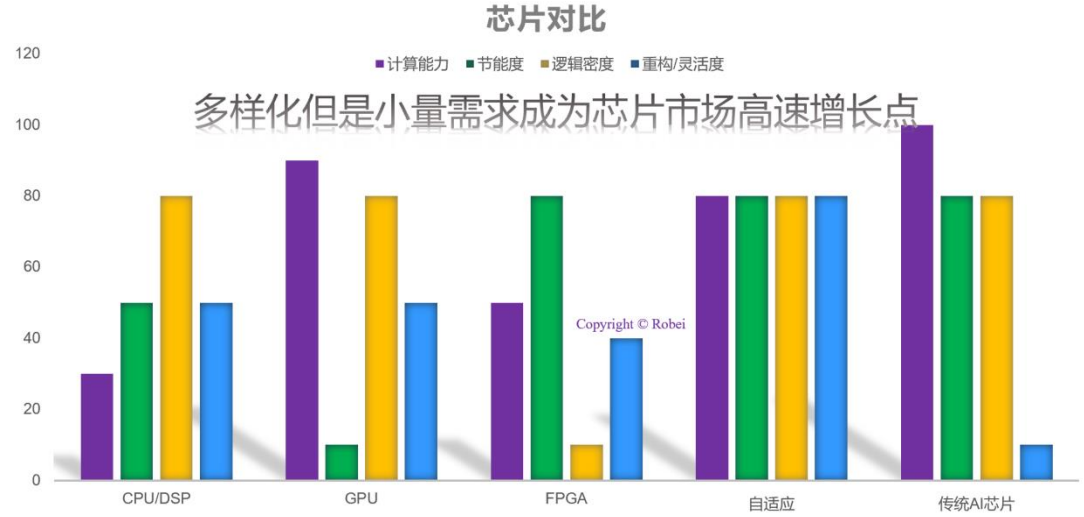


图 17-6 芯片的长项决定了应用特点，短板也限制了其应用领域
（数字仅为示意不代表任何度量标准）

18. 若贝简介

青岛若贝电子有限公司是国内既有 EDA 开发工具又有自主可控芯片的集成电路设计公司。2014 年开始推出的 Robei EDA 工具采用了可视化的方式展现出数字芯片设计中面向对象的设计方法，采用无限分层的设计理念将复杂平铺的集成电路变成可重用、易复用的框图 IP 设计方式。目前 Robei EDA 工具已经在全球 50 多个国家进行使用。

沉芯芯片是若贝用 5 年时间打造的高可靠动态可重构异构芯片，采用自主设计的架构支持 RISC-V 中 RV32IM 指令集，根据芯片型号可以选择 72 核以及 256 核动态可重构阵列。除此以外，芯片还配备了自主研发的 IP 如 QSPI/SPI，PWM，UART，I2C 以及按组可重构 GPIO 等。若贝公司还研发了 Robei IDE 工具，用于沉芯系列芯片的项目开发与编程。

青岛若贝电子有限公司曾获得青岛市科技进步一等奖、教育部春晖杯创新创业大赛优胜奖、山东省泰山产业领军人才、青岛市拔尖人才、青岛市领军人才等。若贝公司组织编写了由电子工业出版社出版的《7 天搞定 FPGA-Robei 与 Xilinx 实战》（ISBN：978-7-121-28310-9）等书籍，并主编了由高教社出版的《数字集成电路设计》（ISBN：978-7-04-053971-4）教材。动态可重构自适应芯片已经获得中国、美国、加拿大的发明专利的授权。未来的若贝，将实现更大的突破，期待和您一起完成！

联系方式：

若贝（无锡）微电子有限公司
地址：中国江苏省无锡市滨湖区建筑路 777 号 A10 楼 206
邮箱: robei@robei.com
网址: http://robei.com
电话: 18562888019

总部：青岛若贝电子有限公司
地址：中国山东省青岛市高新区凯丰国际大厦 A 座 1005。
邮编：266109
电话: 18816653091
邮箱: may@robei.com
网址: http://robei.com