WS2106-2 EC30-EKSTM32 用户手册第二部分

C 扩展开发

CREATE: 2010/06/01
UPDATE: 2010/07/20
Version 1.1
EC30-EKSTM32 Version 1.1
GUTTA Ladder Editor Version 1.1
http://www.plcol.com
http://www.visiblecontrol.com

第 1 章 简介

嵌入式系统和台式机系统之间的一个最大区别就是：大多数嵌入式系统仅仅要求运行一个程序。这个程序在单片机上电的时候开始运行，在断电时停止运行。下面的程序给出在 Cortex-M3 内核中，最简单，也是最常用的一种 C 语言结构：

程序文件：**2-1-1\main.c**：

```
#include "stm32f10x_lib.h"

void x_init(void);
void x_task(void);

void main(void) {
   x_init();
   while(1) {
      x_task();
      }
   }

void x_init(void) {
   }

void x_task(void) {
   }
```

“stm32f10x_lib.h”是 ST 公司提供的 STM32F103 库文件的头文件。STM32F103 库实现了最基本的硬件支持：单片机特殊寄存器的定义以及对外设的基本操作。

main 函数依然是用户 C 语言程序的入口。如程序所示，进入 main 函数后，系统首先调用函数 x_init 来进行必要的初始化。函数 x_init 的工作完成后，系统反复调用任务 x_task，直至系统断电。因为没有操作系统可以返回，我这里需要“超级循环”（就是“死循环”），让系统一直执行 x_task 任务。

假设我们需要实现一个简单的控制：

系统有 3 个输入信号 I0.0、I0.1 和 I0.2；

系统有 2 个输出信号 Q0.0 和 Q0.1。

I0.0 对应一个物理按键为起动信号；

I0.1 对应一个物理按键为停止信号；

I0.2 接故障传感器（超压、过流、过热等信号）的常开触点。

Q0.0 为运行输出，实际驱动一个电机；

Q0.1 为故障输出，实际驱动一个面板故障灯。

系统的控制要求是：

在非故障状态下，按下 I0.0，进入运行状态，Q0.0 接通，电机运行。

在运行状态下，按下 I0.1，进入停止状态，Q0.0 断开，电机停止。

在运行时若发现 I0.2 有信号，进入故障状态。

在故障状态下，电机停止，同时 Q0.1 接通，面板故障灯亮。

若 I0.0、I0.1、I0.2 对应的单片机管脚是 PA0、PA1、PA2，若 Q0.0、Q0.1 对应的单片机管脚是 PB0、PB1，在前面给出的“超级循环”结构中，我们完成 x_init、x_task 这两个函数的定义即可。

程序文件：**2-1-2\main.c**：

```
#include "STM32F103_lib.h"

void x_init(void);
void x_task(void);

int state;

void main(void) {
   x_init();
   while(1) {
      x_task();
      }
   }

void x_init(void) {
   GPIO_InitTypeDef GPIO_InitStructure;

   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB,
ENABLE);

   // I0.0 I0.1 I0.2
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2;
   GPIO_Init(GPIOA, &GPIO_InitStructure);

   // Q0.0 Q0.1
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
   GPIO_Init(GPIOB, &GPIO_InitStructure);

   state = 0;
   }

void x_task(void) {
   if (GPIO_ReadOutputDataBit(GPIOA, GPIO_Pin_2))
      state = 1;
```

```
if (state == 0) {
    if (GPIO_ReadOutputDataBit(GPIOA, GPIO_Pin_0))
        GPIO_SetBits(GPIOB, GPIO_Pin_0);
    if (GPIO_ReadOutputDataBit(GPIOA, GPIO_Pin_1))
        GPIO_ResetBits(GPIOB, GPIO_Pin_0);
} else {
    GPIO_ResetBits(GPIOB, GPIO_Pin_0);
    GPIO_SetBits(GPIOB, GPIO_Pin_1);
    }
}
```

这里我们引入了一个全局变量“int state”用于纪录是否发生过故障。若“state == 0”表示设备处于正常状态；若“state == 1”表示设备处于故障状态。

在 x_init 任务中，函数首先调用“RCC_APB2PeriphClockCmd”来使能 STM32F103 单片机的A、B端口总线。然后调用“GPIO_Init”来具体设置PA0、PA1、PA2的电器特性（10MHz、高阻跟随）；PB0、PB1 的电器特性（10MHz、推勉输出）。最后，初始化全局变量“state = 0”。

在 x_task 任务中，函数首先检查 PA2 端口的电平，若为高电平（I0.2 接收到故障传感器的信号），则设置全局变量“state = 1”。

接下来，根据“state”变量的值，实行不同的控制策略。若“state == 0”，表示没有发生故障，即 PA0 信号可以置位 PB0、PA1 信号可以复位 PB0。若“state != 0”，表示已经发生过故障，此时忽略 PA0 和 PA1，总是复位 PB0（电机停止），同时总是置位 PA2（面板故障灯亮）。

在更加复杂的应用中，x_task 任务往往不会直接读取或者设置端口的状态。而是采取集中读取输入，集中设置输出的办法。系统在内存上建立两个区域，一个是“输入映像区”，一个是“输出映像区”。x_task 任务首先一次性读取全部需要的输入状态（包括模拟量），然后执行具体的控制逻辑。控制逻辑程序不会直接访问硬件端口，而是根据“输入映像区”中内存的值来做逻辑判断，控制逻辑程序也不会直接驱动硬件端口，而是只修改“输出映像区”中内存的值。控制逻辑程序执行结束后，x_task 任务再一次性的设置全部需要输出的状态（包括模拟量）。

这样做最直接的好处是硬件驱动和控制逻辑实现了分离。在外部硬件发生了变化，管脚定义、电平逻辑被修改的情况下，不需要修改控制逻辑，只需要修改读取或者设置映像区域的那部分程序即可。

这样做还有更深层次的原因。虽然目前单片机的速度已经足够快了（STM32F103 可以达到 72MHz），但是这个速度依然慢于现实世界的物理响应。在逻辑程序执行的过程中，输入端口的值很有可能发生变化，输出端口的变化同时也可能造成输入端口发生变化。

逻辑控制程序的编写如果考虑上述问题，会造成程序状态和实际的设备状态存在紧密耦合。我们期望逻辑控制程序应该是简单的根据输入得到输出，在一次扫描的过程中，输入是静态不变的。得到的输出也不需要立即实现，一方面这个输出可能还不是最终结果（例如还未加入互锁的判断），另一方面这些输出可能又会改变输入。

基于集中扫描输入、执行控制逻辑、集中执行输出的思想，可以将上述常用的 C 语言结构修改成下面的样子：

程序文件：**2-1-3\main.c**：

```
#include "STM32F103_lib.h"

void x_init(void);
void x_get_input(void);
void x_logic_scan(void);
void x_set_output(void);

int img_input = 0;
int img_output = 0;

void main(void) {
    x_init();
    while(1) {
        x_get_input();
        x_logic_scan();
        x_set_output();
        }
    }

void x_init(void) {
    }

void x_get_input(void) {
    }

void x_logic_scan(void) {
    }

void x_set_output(void) {
    }
```

main 循环中的 x_task 任务被拆成了 3 个任务 x_get_input、x_logic_scan、x_set_output，分别用于读取输入端口、执行控制逻辑、设置输出端口。同时，创建了两个全局变量“int img_input = 0”和“int img_output = 0”，用于保存端口数据，即作为“输入映象区”和“输出映象区”。基于这个结构，可以将最开始的例子调整如下：

程序文件：**2-1-4\main.c**：

```
#include "STM32F103_lib.h"

void x_init(void);
void x_get_input(void);
void x_logic_scan(void);
void x_set_output(void);
```

```
int img_input = 0;
int img_output = 0;

int state;

void main(void) {
    x_init();
    while(1) {
        x_get_input();
        x_logic_scan();
        x_set_output();
        }
    }

void x_init(void) {
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB,
ENABLE);

    // I0.0 I0.1 I0.2
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // Q0.0 Q0.1
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    state = 0;
    }

void x_get_input(void) {
    if (GPIO_ReadOutputDataBit(GPIOA, GPIO_Pin_0))
        img_input |= GPIO_Pin_0;
    else
        img_input &= ~GPIO_Pin_0;

    if (GPIO_ReadOutputDataBit(GPIOA, GPIO_Pin_1))
        img_input |= GPIO_Pin_1;
```

```
    else
        img_input &= ~GPIO_Pin_1;

    if (GPIO_ReadOutputDataBit(GPIOA, GPIO_Pin_2))
        img_input |= GPIO_Pin_2;
    else
        img_input &= ~GPIO_Pin_2;
    }

void x_logic_scan(void) {
    if (img_input & GPIO_Pin_2)
        state = 1;

    if (state == 0) {
        if (img_input & GPIO_Pin_0)
            img_output |= GPIO_Pin_0;
        if (img_input & GPIO_Pin_1)
            img_output &= ~GPIO_Pin_0;
    } else {
        img_output &= ~GPIO_Pin_0;
        img_output |= GPIO_Pin_1;
        }
    }

void x_set_output(void) {
    if (img_output & GPIO_Pin_0)
        GPIO_SetBits(GPIOB, GPIO_Pin_0);
    else
        GPIO_ResetBits(GPIOB, GPIO_Pin_0);

    if (img_output & GPIO_Pin_1)
        GPIO_SetBits(GPIOB, GPIO_Pin_1);
    else
        GPIO_ResetBits(GPIOB, GPIO_Pin_1);
    }
```

EC30-EKSTM32 是一个完整的 PLC 系统。用户可以直接使用指令表（STL）或者是梯形图（LAD）来编写控制逻辑。同时，STM32F103 作为 PLC 使用时，常用的一些硬件驱动可以在 GUTTA Ladder Editor 软件中自由配置（例如 I/O 分配、A/D 转换、I^2C 通讯、高数计数和高数脉冲输出等），详细的配置方法请参考本手册的第一部分。如果这些硬件驱动通过配置后依然不能满足实际的需求，您也可以用 C 语言来开发适合自己硬件的驱动。除了硬件方面的驱动，您也可以用 C 语言来开发特殊的 PLC 指令，甚至是直接使用 C 语言完成部分控制逻辑。下面将展示如何在 EC30-EKSTM32 这个 PLC 系统上使用 C 语言开发自己的硬件驱动和控制逻辑。为了简单起见，我们延续本节开始提出的控制要求。为了能够让程序真

正在 EC30-EKSTM32 开发板上运行和调试，根据电路图我们知道，开发板上 I0.0、I0.1、I0.2 对应的管脚为 PC8、PC6、PC7，开发板上 Q0.0、Q0.1 对应的管脚为 PB8、PB9。下面的程序在管脚的驱动做了相应的修改。

程序文件：**2-1-5\main.c**：

```
#include "system.h"
#include "STM32F103_lib.h"

void x_init(void);
void x_get_input(void);
void x_logic_scan(void);
void x_set_output(void);

uint32_t main(uint32_t action, uint32_t param) {

   switch (action) {
      case CUSTOM_MODULE_RESET:
         x_init();
         return RT_OK;

      case CUSTOM_IO_GET_INPUT:
         x_get_input();
         return RT_OK;

      case CUSTOM_IO_SET_OUTPUT:
         x_set_output();
         return RT_OK;

      case CUSTOM_LGC_INT_LEAVE:
         if (param == 0) {
            x_logic_scan();
            }
         return RT_OK;
      }

   return RT_CANCEL;
   }

void x_init(void) {
   GPIO_InitTypeDef GPIO_InitStructure;

   // I0.0 I0.1 I0.2
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
```

```
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    // Q0.0 Q0.1
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    }

void x_get_input(void) {
    if (!GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_8))
        IB(0) |= (1<<0);
    else
        IB(0) &= ~(1<<0);

    if (!GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_6))
        IB(0) |= (1<<1);
    else
        IB(0) &= ~(1<<1);

    if (!GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_7))
        IB(0) |= (1<<2);
    else
        IB(0) &= ~(1<<2);
    }

void x_logic_scan(void) {
    if (IB(0) & (1<<2))
        MB(0) = 1;

    if (MB(0) == 0) {
        if (IB(0) & (1<<0))
            QB(0) |= (1<<0);
        if (IB(0) & (1<<1))
            QB(0) &= ~(1<<0);
    } else {
        QB(0) &= ~(1<<0);
        QB(0) |= (1<<1);
        }
    }

void x_set_output(void) {
    if (QB(0) & (1<<0))
```

```
    GPIO_ResetBits(GPIOB, GPIO_Pin_8);
else
    GPIO_SetBits(GPIOB, GPIO_Pin_8);

if (QB(0) & (1<<1))
    GPIO_ResetBits(GPIOB, GPIO_Pin_9);
else
    GPIO_SetBits(GPIOB, GPIO_Pin_9);

++ MB(20);
}
```

对比 **2-1-4\main.c** 的 C 程序，我们可以发现下面一些有趣的改动。

首先，添加了一个新的头文件“system.h”，这个文件包含了 PLC 系统的一些常数宏定义，以及一些 PLC 变量宏定义。例如后面使用到的“MB(0)”就是一个预定义宏，这个宏实际引用的内存就是 PLC 变量“MB0”。

main 函数的结构发生了重大的变化：

“**uint32_t** main(**uint32_t** action, **uint32_t** param)”

main 函数调用有两个参数，同时有一个返回值。观察 main 函数的具体实现可以发现，这里没有“超级循环”，main 函数根据 action 参数的不同，分别调用 x_init、x_get_input、x_set_output、x_logic_scan 这 4 个函数，调用结束后，立即返回 RT_OK。如果 action 参数不能触发上述任何一个函数调用，则返回 RT_CANCEL。

和普通 C 语言工程不同，这段程序在实际编译的时候不能够使用编译器的默认工程模板，而必须使用特别提供的工程模板（对 SRAM 和 FLASH 的使用范围做了特别的定义，并修改了默认的启动文件以支持这种特殊的 main 函数格式）。这个工程模板可以在手册的附件中找到。编译链接成功后，编译器会生成一个 HEX 文件。这个文件的下载不需要特别的工具，EC30-EKSTM32 系统直接支持 MODBUS 协议，并在 MODBUS 协议上扩展了 GUTTA 监控和传输协议，其中也包括用户扩展 C 语言二进制程序的下载。只要 PLC 核位于正常工作状态，就可以使用 GUTTA Flash Utility 工具进行下载。

GUTTA Flash Utility 软件运行的界面如图所示，这里我们设置工作模式为“写入用户程序（HEX 格式）”。通过点击“HEX 文件”按钮，选择工程生成的 HEX 文件。然后在下拉框中选择 PLC 型号为 EC30-EKSTM32。点击“通讯设置”按钮配置实际和 EC30-EKSTM32 通讯的电脑端口。这些准备工作都完成后，便可点击“写入 FLASH”按钮，开始下载用户 C 语言程序。

为了防止 PLC 逻辑和用户 C 语言的逻辑发生冲突，最好在 GUTTA Ladder Editor 中新建一个不包含任何硬件配置的空项目，将此项目下载到 PLC 核中。这时便可通过改变学习板上 I0.0、I0.1、I0.2 的按键状态，观察 Q0.0、Q0.1 的动作。

上面的 main 函数根据 action 参数的不同，分别调用 x_init、x_get_input、x_set_output、x_logic_scan 这 4 个函数。调用结束后，立即返回 PLC 系统。EC30-EKSTM32 的 PLC 系统在自己运行的同时，会在恰当的时候调用用户 C 语言程序。

例如 PLC 系统在 PLC 复位并完成自己的硬件配置之后（系统块配置），将尝试向用户 main 发送“CUSTOM_MODULE_RESET”消息，告诉用户，你可以初始化自己的硬件了。

PLC 系统的工作依然采用集中输入、逻辑控制、集中输出的模式。PLC 系统完成自己的默认输入采集后，便向用户发送“CUSTOM_IO_GET_INPUT”消息，告诉用户，你可以采集外部信号并将结果写入输入映象区了。

“CUSTOM_IO_GET_INPUT”消息返回后，PLC 系统开始自己的主循环扫描，即指令表（STL）或梯形图（LAD）的控制逻辑，主循环扫描结束后，便向用户发送“CUSTOM_LGC_INT_LEAVE”消息，参数“param == 0”表示为 INT0 程序，也就是 PLC 系统的主循环扫描（MAIN_SCAN）。这里我们在“CUSTOM_LGC_INT_LEAVE”消息响应中用 C 语言编写自己的控制逻辑（即 x_logic_scan），作为主循环扫描（MAIN_SCAN）的补充。

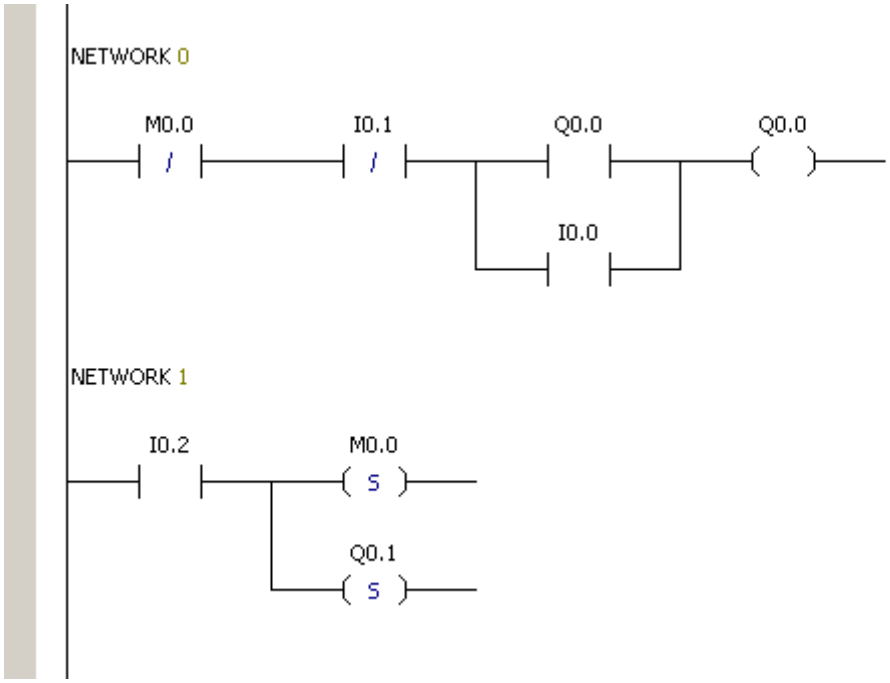
我们完全可以让 PLC 系统自己来实现 x_logic_scan 的控制逻辑，使用 PLC 系统自己的指令表（STL）或者是梯形图（LAD）写这种位控制逻辑更加方便。去掉 x_logic_scan 的实现，上述程序应该是这样子：

程序文件：**2-1-6\main.c**：

```
#include "system.h"
#include "STM32F103_lib.h"

void x_init(void);
void x_get_input(void);
void x_set_output(void);

uint32_t main(uint32_t action, uint32_t param) {

    switch (action) {
        case CUSTOM_MODULE_RESET:
            x_init();
            return RT_OK;

        case CUSTOM_IO_GET_INPUT:
            x_get_input();
            return RT_OK;

        case CUSTOM_IO_SET_OUTPUT:
            x_set_output();
            return RT_OK;

        case CUSTOM_LGC_INT_LEAVE:
            return RT_OK;
        }

    return RT_CANCEL;
    }

void x_init(void) {
    GPIO_InitTypeDef GPIO_InitStructure;

    // I0.0 I0.1 I0.2
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    // Q0.0 Q0.1
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9;
```

```
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    }

void x_get_input(void) {
    if (!GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_8))
        IB(0) |= (1<<0);
    else
        IB(0) &= ~(1<<0);

    if (!GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_6))
        IB(0) |= (1<<1);
    else
        IB(0) &= ~(1<<1);

    if (!GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_7))
        IB(0) |= (1<<2);
    else
        IB(0) &= ~(1<<2);
    }

void x_set_output(void) {
    if (QB(0) & (1<<0))
        GPIO_ResetBits(GPIOB, GPIO_Pin_8);
    else
        GPIO_SetBits(GPIOB, GPIO_Pin_8);

    if (QB(0) & (1<<1))
        GPIO_ResetBits(GPIOB, GPIO_Pin_9);
    else
        GPIO_SetBits(GPIOB, GPIO_Pin_9);
    }
```

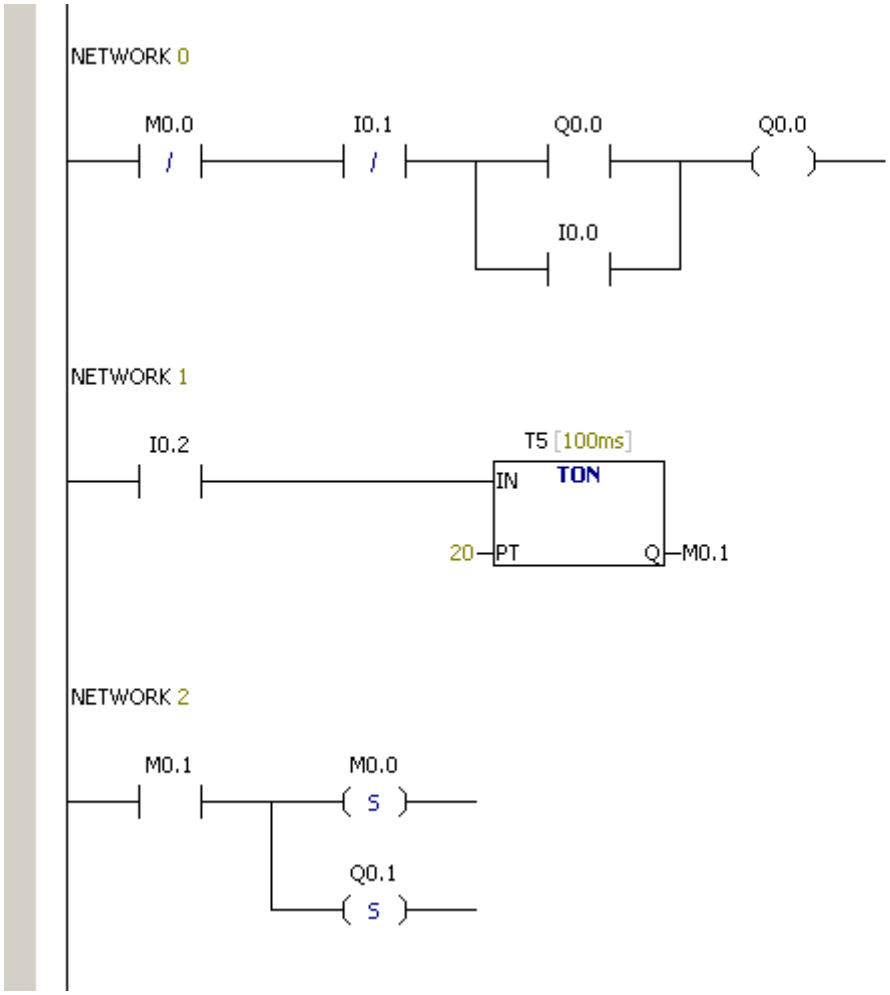
程序除了没有 x_logic_scan 函数的声明和定义之外，几乎没有其他任何变化。同样的，利用 GUTTA Flash Utility 将编译链接成功的 HEX 文件下载到 EC30-EKSTM32。然后在 GUTTA Ladder Editor 中编写如下程序并下载到 EC30-EKSTM32。



这段梯形图和前面我们用 C 语言编写的 x_logic_scan 函数实现几乎等效。Q0.0 的控制采用了经典的起跑停控制电路。故障信号 I0.2 会同时置位故障标志 M0.0 和面板输出 Q0.1，同时 M0.0 会切断并 Q0.0 的输出。

假设我们这里 I0.2 实际代表的是超压信号（压力过高）。在真实的世界中，压力瞬间过高可能是因为其他原因，例如检测管路发生了物理撞击。如果压力检测的对象为不可压缩的液体，压力会发生瞬间的振荡，而这种振荡很有可能错误的设置故障标志而关闭整个设备。为了防止这种错误的操作，可以采用对故障信号 I0.2 进行延时判断的方法。

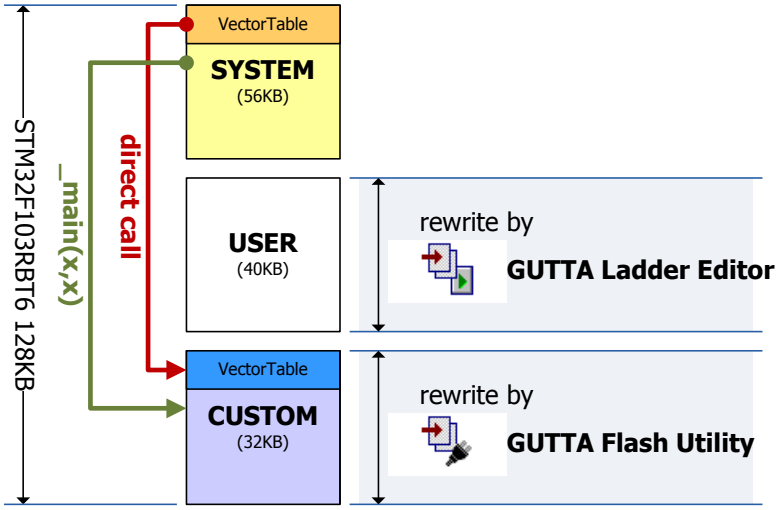
程序文件：**2-1-6\ main.vcw**：



在 C 语言的 x_logic_scan 中，实现一个延时判断是很麻烦的，而在梯形图中，只需要简单的调用 TON 指令即可。

第 2 章 消息机制

上一节我们通过一个实例，展示了 PLC 系统和用户 C 语言结合的若干细节。这里我们做一个更加系统完整的介绍。首先，我们来看 EC30-EKSTM32 对 STM32F103RBT6 的 FLASH 是如何规划的。

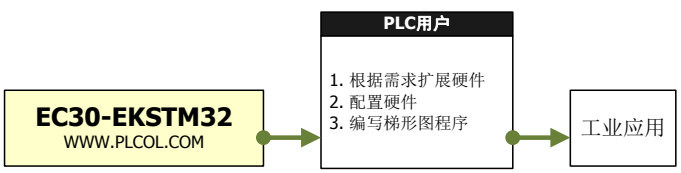


STM32F103RBT6 的 128K FLASH 被划分成 3 大部分。

1. SYSTEM 区是 PLC 系统区，系统区在 EC30-EKSTM32 出厂时被一次性写入，以后不能改变。系统区的程序主要负责 PLC 的通讯协议，PLC 梯形图逻辑的解析，同时需要根据系统块数据来配置 STM32F103 的外围硬件。SYSTEM 区上的程序，在适当的时机，会调用 CUSTOM 区的_main 函数，_main 函数经过一定的消息预处理后，最终调用 C 语言二次开发的用户 main。STM32F103 的中断向量地址从 SYSTEM 区的 VectorTable 开始，SYSTEM 区的中断服务程序根据 NIVC 模块的配置，有选择的将中断服务转发到 CUSTOM 的 VectorTable 中去处理。
2. USER 区是 PLC 程序区，用于存放用户的指令表（STL）或梯形图（LAD）。这部分区域由 GUTTA Ladder Editor 软件维护。在 GUTTA Ladder Editor 中，你可以编写梯形图程序并下载到 USER 区，或者是从现有的 USER 区将梯形图程序上传到 GUTTA Ladder Editor 软件中。
3. CUSTOM 区是 C 语言二次开发区。对 PLC 系统的扩展开发全部位于这个区。上一章中用户 main 就是通过 GUTTA Flash Utility 软件下载到这个区域的。这个区域的程序不提供上传机制。根据上图我们可以知道，使用 GUTTA Ladder Editor 软件更新 PLC 梯形图程序时，并不会修改 CUSTOM 区的任何数据。CUSTOM 区域中的程序可作为实际 PLC 产品的一部分随 PLC 硬件发布。

下面我们来看看 EC30-EKSTM32 核的几种基本使用形式。

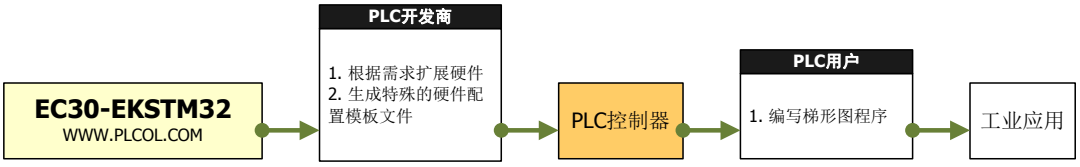
1. PLC 控制器自用。



这种模式为 PLC 系统自用。例如某设备制造商买来 EC30-EKSTM32 内核，根据自己的

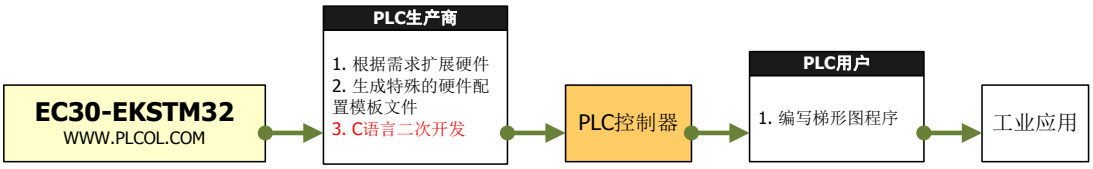
应用设计一套简单的外围硬件，采购必需的电子元器件进行组装后，直接将产品应用于自己的设备之上。硬件可以通过软件进行配置，控制逻辑可以使用梯形图编程，并且具有标准的在线监控和调试接口。使用 EC30-EKSTM32 大大的降低了设备制造商开发控制器产品的难度。在设备型号比较多样，用户定制化需求比较多的场合，梯形图方式的逻辑编写也极大的提高了开发的速度。

2. PLC 控制器作为专业产品。



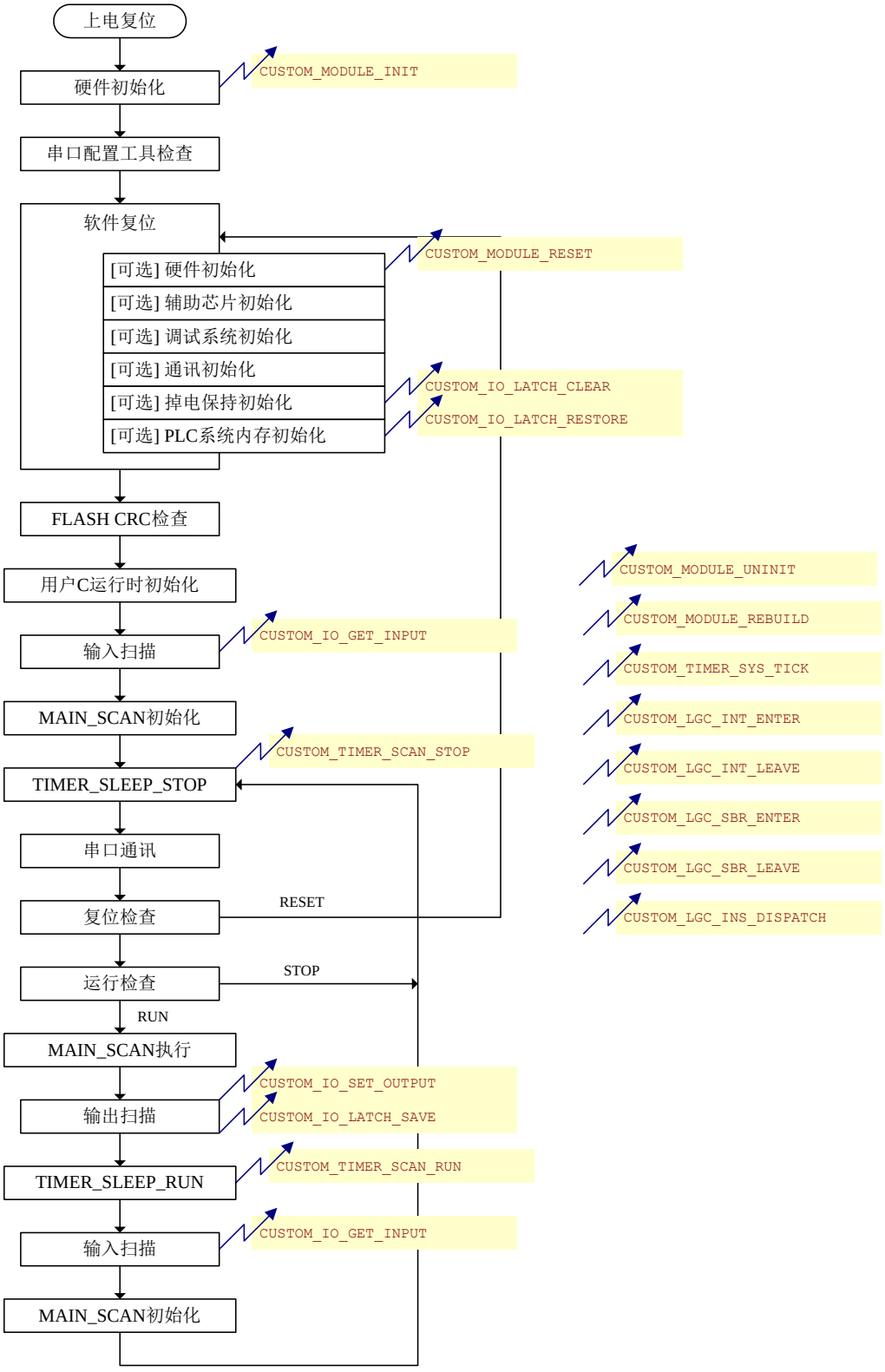
这种模式下，PLC 控制器作为专业产品独立销售。例如 PLC 生产商买来 EC30-EKSTM32 内核，根据自己对行业的理解，开发出一套针对行业应用的 PLC 控制器。PLC 生产商必须生成一个与其硬件配套的模板文件，同时修改 GUTTA Ladder Editor 配置文件，将不希望 PLC 用户看到的配置细节隐藏。将这个特殊的配置文件和 GUTTA Ladder Editor 软件随 PLC 控制器一起发布给 PLC 用户。PLC 用户每新建一个工程，都将自动采用这个特殊的硬件配置文件。

3. PLC 控制器作为专业产品，进行了二次开发。



一些特殊的应用需要特殊的外围硬件，而 GUTTA Ladder Editor 软件中有限的几个驱动肯定不能涵盖所有需求。这个时候，就需要 PLC 生产商在 EC30-EKSTM32 核的基础上进行 C 语言二次开发，并将二次开发程序写入 CUSTOM 区随 PLC 硬件一起发布。

二次开发的关键点就在于，响应 PLC 系统发送给用户 main 的各种消息。PLC 系统在整个运行流程中，会在很多地方向用户 main 发送消息。一个定制化的 PLC 系统，关键在于用户 main 如何响应这些消息。我们首先需要了解整个 PLC 系统的工作流程，以便理解每个消息的具体含义，



- 硬件初始化

  STM32F103 上电复位后，首先进行硬件初始化。硬件初始化完成后立即发送 CUSTOM_MODULE_INIT 消息。不建议用户在这个消息中初始化自己的硬件，这是因为在随后软件复位中，绝大部分硬件又被初始化为 STM32F103 的默认值。PLC 系统只在这里初始化很小一部分配置不变的硬件，为的是使 STM32F103 的串口可以工作。

- 串口配置工具检查

  串口 PLC 系统会等待 200ms 看是否有 PLC 启动配置工具（GUTTA Flash Utility）向单片机发送配置信息，若没有，进入软件复位状态。

- 软件复位

  软件复位有 6 个可选的工作部分。具体是否执行某个工作部分取决于复位源：

  上电复位、通讯强制复位、调试模式通讯强制复位。

  硬件初始化部分初始化完成后立即发送 CUSTOM_MODULE_RESET 消息。用户自己的硬件初始化，建议在这个消息响应中完成。掉电保持初始化会向用户发送 CUSTOM_LATCH_CLEAR 消息，这个消息表示需要清除所有的掉电保持数据。只有在 PLC 的复位源为通讯强制复位时，才会执行这个部分（有新程序下载完成，需要清除已有的掉电保持数据）。PLC 系统内存初始化会在最后向用户发送 CUSTOM_LATCH_RESTORE 消息，表示将掉电保持的数据恢复到 PLC 内存中去。

- FLASH CRC 检查

  进行 FLASH 的 CRC 校验，如果校验不成功设置错误号。

- 用户 C 运行时初始化
- 输入扫描

  进行一次输入扫描并向用户发送 CUSTOM_IO_GET_INPUT 消息，告诉用户，可以向输入映像区填写数据了。

- MAIN_SCAN 初始化
- TIMER_SLEEP_STOP

  从 TIMER_SLEEP_STOP 开始，便进入了单片机的“超级循环”。每执行一次循环，相当于 PLC 系统完成了一个扫描周期。

  TIMER_SLEEP_STOP 操作只在 PLC 模拟器中有实际内容，但总是向用户发送 CUSTOM_TIMER_SCAN_STOP 消息。用户程序如果需要往超级循环中添加程序，响应 CUSTOM_TIMER_SCAN_STOP 消息即可。

- 串口通讯
- 复位检查

  如果发现有复位源存在，跳回软件复位。

- 运行检查

  如果 PLC 位于停止状态，跳回 TIMER_SLEEP_STOP。

- MAIN_SCAN 执行

  执行 MAIN_SCAN 用户梯形图逻辑（也既 INT0）。

  进入前会向用户发送 CUSTOM_LGC_INT_ENTER 消息。

  退出后会向用户发送 CUSTOM_LGC_INT_LEAVE 消息。

- 输出扫描

  进行一次输出扫描并向用户发送 CUSTOM_IO_SET_OUTPUT 消息，告诉用户，可以将输出映像中的数据更新到实际的硬件设备中去了。同时，发送 CUSTOM_LATCH_SAVE 消息，告诉用户，可以将需要掉电保持的数据保存到掉电保持外设中去。

- TIMER_SLEEP_RUN

  TIMER_SLEEP_RUN 操作只在 PLC 模拟器中有实际内容，但总是向用户发送 CUSTOM_TIMER_SCAN_RUN 消息。用户程序如果需要往超级循环中添加程序，响应 CUSTOM_TIMER_SCAN_RUN 消息即可。

  和 CUSTOM_TIMER_SCAN_STOP 消息不同，CUSTOM_TIMER_SCAN_RUN 只在

PLC 运行时发送，而 CUSTOM_TIMER_SCAN_STOP 在任何时候都会发送。

- 输入扫描

  进行一次输入扫描并向用户发送 CUSTOM_IO_GET_INPUT 消息，告诉用户，可以向输入映像区填写数据了。

- MAIN_SCAN 初始化

  执行 MAIN_SCAN 的初始化工作，无条件跳回 TIMER_SLEEP_STOP。

目前 EC30-EKSTM32 支持的消息：

消息	值	说明
CUSTOM_MODULE_INIT	0001H	模块初始化
CUSTOM_MODULE_UNINIT	0002H	模块释放
CUSTOM_MODULE_RESET	0003H	模块复位
CUSTOM_MODULE_REBUILD	0004H	模块重建（若模块需要修改 FLASH 数据）
CUSTOM_IO_GET_INPUT	0005H	输入扫描
CUSTOM_IO_SET_OUTPUT	0006H	输出扫描
CUSTOM_IO_LATCH_CLEAR	0007H	掉电保持数据清除
CUSTOM_IO_LATCH_SAVE	0008H	掉电保持数据保存
CUSTOM_IO_LATCH_RESTORE	0009H	掉电保持数据恢复
CUSTOM_TIMER_SCAN_RUN	000AH	循环扫描（只在 PLC 运行时）
CUSTOM_TIMER_SCAN_STOP	000BH	循环扫描（任何时候）
CUSTOM_TIMER_SYS_TICK	000CH	系统 1ms 节拍
CUSTOM_LGC_INT_ENTER	000DH	进入 INT 调用
CUSTOM_LGC_INT_LEAVE	000EH	退出 INT 调用
CUSTOM_LGC_SBR_ENTER	000FH	进入 SBR 调用
CUSTOM_LGC_SBR_LEAVE	0010H	推出 SBR 调用
CUSTOM_LGC_INS_DISPATCH	0011H	用户自定义指令派发

第 3 章 用户 C 运行时

我们知道，嵌入式开发中，C 语言的 main 函数并不是单片机的上电复位入口。单片机在进入 main 函数之前，一般还需要运行一段所谓的启动程序。启动程序需要完成一些基础的配置工作。其主要内容有：

- 基础的硬件配置（例如配置扩展 SDRAM）。
- 初始化 C 语言的栈，设置栈指针。
- 初始化 C 语言中无初始化值的全局变量（一般设置为 0）。
- 初始化 C 语言中有初始化值的全局变量。
- 全局对象的构造函数（只在 C++中需要）。

编译器一般会提供一个 C 运行库以完成上述操作。单片机的上电复位向量一般指向库中的某个入口函数，这个函数在完成上述操作后，才开始调用用户编写的 main 函数。这个函数同时也不期望 main 函数返回，若 main 函数真的返回了，由于没有操作系统可以返回，简单的以一个死循环等候之。

在 EC30-EKSTM32 的 C 语言系统中，main 函数响应某个消息后，必须立即返回，否则会诸塞 PLC 系统的正常运行。为了支持这种特殊的 main 函数格式，我们提供的工程模板文件除了在空间上（FLASH、SRAM）修改了默认值之外，还必须提供一个特殊的启动程序。

这个启动程序必须在正确的时候初始化用户 C 语言环境。

程序文件：**2-3-1\main.c**：

```
#include "system.h"
#include "STM32F103_lib.h"
#include <string.h>

char g[] = "www.plcol.com";

uint32_t main(uint32_t action, uint32_t param) {

    switch (action) {
        case CUSTOM_IO_GET_INPUT:
            strcpy((char*)&MB(0), g);
            return RT_OK;
        }

    return RT_CANCEL;
    }
```

这里，我们创建了一个必须初始化的变量“`char g[] = "www.plcol.com"`”。为了确定这个变量被正确初始化，在 **CUSTOM_IO_GET_INPUT** 消息中，我们调用“`strcpy((char*)&MB(0), g)`”将这个变量的值拷贝到 MB0 开始的内存中去。将这段程序的 HEX 文件用 GUTTA Flash Utility 下载到 EC30-EKSTM32 中去，然后用 GUTTA Ladder Editor 连线，观察 MB0 ~ MB15 的值：

状态表

	地址	数据类型	值	强制
	MB0	SINT	119	
	MB1	SINT	119	
	MB2	SINT	119	
	MB3	SINT	46	
	MB4	SINT	112	
	MB5	SINT	108	
	MB6	SINT	99	
	MB7	SINT	111	
	MB8	SINT	108	
	MB9	SINT	46	
	MB10	SINT	99	
	MB11	SINT	111	
	MB12	SINT	109	
	MB13	SINT	0	
	MB14	SINT	0	
	MB15	SINT	0	

GUTTA Ladder Editor 不能够直接显示字符串，但是可以通过数值判断出，MB0 ~ MB13 被正确的拷贝为“`www.plcol.com`”。

第 4 章 添加 PLC 指令

某些行业应用可能需要特殊的数学算法，例如改进过的 PID 运算、传感器数据处理等。有些控制方式特别是数学算法用梯形图来实现并不是特别的方便，运行效率上也有一些折扣。在针对行业应用进行 PLC 的二次开发中，给用户提供一套针对行业特点的特殊 PLC 指令，对最终用户的 PLC 编程将是非常有帮助的。

出于简单的考虑，这里我们展示如何添加一条 DIV_IR 指令到 EC30-EKSTM32 中去。和标准的除法指令不同，DIV_IR 接受两个 16 位有符号整数，将这两个整数相除后，将商存放在第 1 个输出操作数中，将余数存放在第 2 个输出操作数中。

1 修改 ManagerFun.XML 文件

在 GUTTA Ladder Editor 软件的安装文件夹中，有一个名为 GuttaLad 的子文件夹。在 GuttaLad 的子文件夹中，又存在若干子文件夹，其中每一个子文件夹代表一种 CPU 配置。打开 EC30-EKSTM32 这个文件夹，会看到一个 ManagerFun.XML 文件。ManagerFun.XML 文件定义了当前 CPU 类型下所有指令的名称、格式、以及转换规则。ManagerFun.XML 文件格式的详细描述可以参考文档《UM4003 指令描述文件规范》。根据这个规范，结合我们需要添加指令的要求，在 ManagerFun.XML 文件中添加以下 XML 程序：

- 中间指令（ORG）的描述部分：

```
<FunOrg>
  <Unit Name="/IR" Comment="" Parent="" Type="Output" Node="1">
    <Operand>
      <FunOperand Name="" Capacity="Wcvp:" Const="Int" />
      <FunOperand Name="" Capacity="Wcvp:" Const="Int" />
      <FunOperand Name="" Capacity="Wv:" Const="Int" />
      <FunOperand Name="" Capacity="Wv:" Const="Int" />
    </Operand>
  </Unit>
</FunOrg>
```

- 指令表指令（STL）的描述部分：

```
<FunStl>
  <Unit Name="/IR" Comment="${Divide Integer Get Remainder}${带余数整数除法}"
Parent="${Custom}${用户}" Slot="224">
    <Operand>
      <FunOperand Name="IN1" Capacity="Wcvp:" Const="Int" />
      <FunOperand Name="IN2" Capacity="Wcvp:" Const="Int" />
      <FunOperand Name="OUT1" Capacity="Wv:" Const="Int" />
      <FunOperand Name="OUT2" Capacity="Wv:" Const="Int" />
    </Operand>
  </Unit>
</FunStl>
```

- 梯形图指令（LAD）的描述部分：

```
<FunLad>
```

```
    <Unit Name="/IR" Comment="${Divide Integer Get Remainder}${带余数整数除法}"
Parent="${Custom}${用户}" Look="Block" Keyword="DIV_IR">
      <PowerIn>
        <FunPower Name="EN" />
      </PowerIn>
      <OperandIn>
        <FunOperand Name="IN1" Capacity="Wcvp:" Const="Int" />
        <FunOperand Name="IN2" Capacity="Wcvp:" Const="Int" />
      </OperandIn>
      <OperandOut>
        <FunOperand Name="OUT1" Capacity="Wv:" Const="Int" />
        <FunOperand Name="OUT2" Capacity="Wv:" Const="Int" />
      </OperandOut>
    </Unit>
  </FunLad>
```

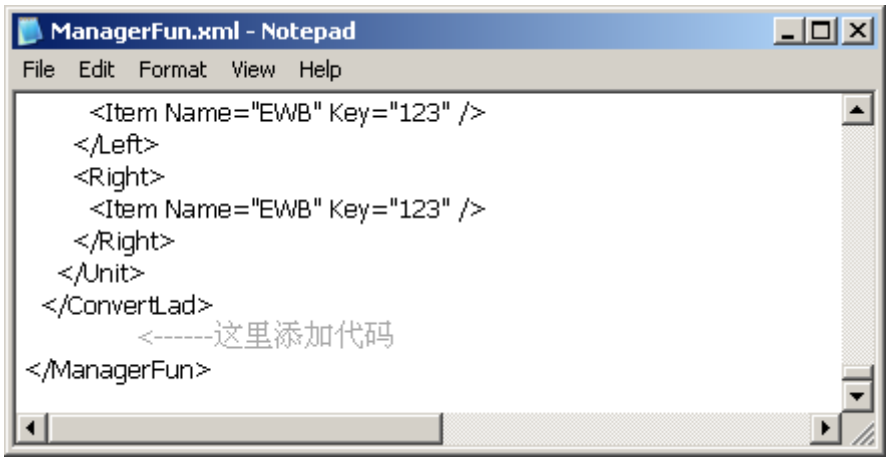
- 转换规则（STL←→ORG）的描述部分：

```
  <ConvertStl>
    <Unit>
      <Left>
        <Item Name="/IR" Key="1234" />
      </Left>
      <Right>
        <Item Name="/IR" Key="1234" />
      </Right>
    </Unit>
  </ConvertStl>
```

- 转换规则（LAD←→ORG）的描述部分：

```
  <ConvertLad>
    <Unit>
      <Left>
        <Item Name="/IR" Key="1234" />
      </Left>
      <Right>
        <Item Name="/IR" Key="1234" />
      </Right>
    </Unit>
  </ConvertLad>
```

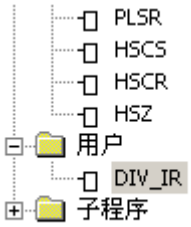
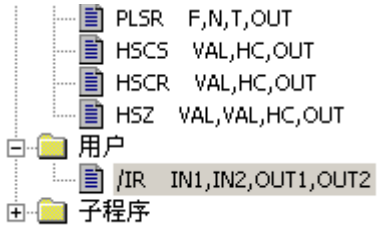
将上面 5 段程序添加到 ManagerFun 节点的最后面：



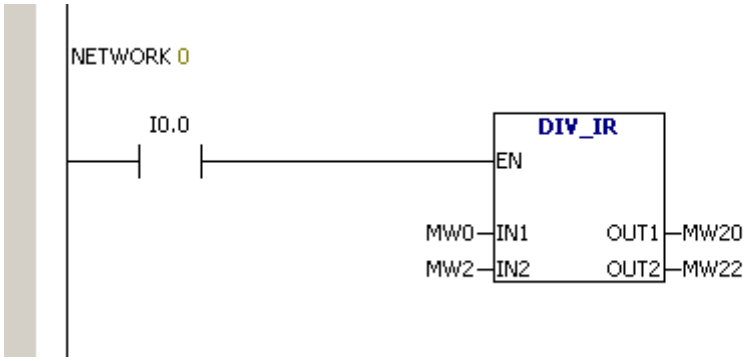
保存文件后，便完成了 ManagerFun.xml 文件的修改。修改后的 ManagerFun.xml 文件可以在 **UserProjectForIAR.zip/UserProjectForKeil.zip** 压缩包中找到。

2 验证 ManagerFun.XML 文件

重新运行 GUTTA Ladder Editor 软件，检查在 STL 编辑模式和 LAD 编辑模式下，是否能够找到刚才加入的/IR 指令：

LAD 编辑模式	STL 编辑模式
	

然后基于/IR 指令编写一个简单的梯形图（LAD）程序：



看是否能够顺利的转化成指令表（STL）格式：

```
NETWORK 0

 LD I0.0
 /IR MW0, MW2, MW20, MW22
```

如果两种模式下指令的编辑和相互转化都没有问题，则说明 ManagerFun.XML 文件中关于/IR 指令的描述没有问题。

3 编写用户 main 函数并下载

PLC 解释系统在解析 PLC 程序的时候，若发现指令号大于或等于 224，则立即向用户 main 函数发送 **CUSTOM_LGC_INS_DISPATCH** 消息。由于指令号最大只能为 255，故用户最多可以在 main 函数中实现 224 ~ 255 即 32 条指令。下面给出/IR 指令的具体实现。

程序文件：**2-4-1\ main.c**:

```
#include "system.h"
#include "STM32F103_lib.h"

void excute_DIV_IR(void);

uint32_t main(uint32_t action, uint32_t param) {

    switch (action) {
        case CUSTOM_LGC_INS_DISPATCH:
            switch (param) {
                case 224:
                    excute_DIV_IR();
                    return RT_OK;
                }
        break;
        }

    return RT_CANCEL;
    }

void excute_DIV_IR(void) {
    if (theResState->itStackData & 1) {
        AS16(2) = AS16(0) / AS16(1);
        AS16(3) = AS16(0) % AS16(1);
        }
    }
```

main 函数只处理 **CUSTOM_LGC_INS_DISPATCH** 消息，且仅当“`param == 224`”时，调用 excute_DIV_IR 函数且返回 RT_OK。在 excute_DIV_IR 中，如果发现数据栈栈顶为 1（“`theResState->itStackData & 1`”），就执行除法操作。AS16 是一个宏，用于得到 PLC 指令的操作数，我们看看在“`system.h`”中，关于宏 AS16 的定义：

```
    #define AP(x)           (theResPar[x].itPtr)
    #define AB(x)           (theResPar[x].itBit)
    #define AS8(x)          (*(int8_t*)AP(x))
    #define AS16(x)         (*(int16_t*)AP(x))
    #define AS32(x)         (*(int32_t*)AP(x))
```

```
#define AU8(x)          (*(uint8_t*)AP(x))
#define AU16(x)         (*(uint16_t*)AP(x))
#define AU32(x)         (*(uint32_t*)AP(x))
```

其中“AP(x)”宏表示第 x 个操作数的指针。在 EC30-EKSTM32 中，x 的范围可以是 0 ~ 7。“AS16(x)”宏表示第 x 个操作数的有符号 16 位整型数据。

4 测试

将这个项目生成的 HEX 文件用 GUTTA Flash Utility 软件下载到 EC30-EKSTM32 中，并将最开始的梯形图程序通过 GUTTA Ladder Editor 下载到 EC30-EKSTM32 中。

强制 I0.0 为 1，将 200 写入 MW0，30 写入 MW2，观察结果的变化：

程序文件：**2-4-1\ main.vcw**：