AN2102 如何在 GUTTA 编译系统中添加自定义指令

COPYRIGHT © 2008 WWW.VISIBLECONTROL.COM

2008/11/25

概述..2
具体步骤..2
第一步：确定指令格式，绘制图形..2
第二步：修改*ManagerFun.xml*文件..3
找到*ManagerFun.xml*文件、打开并编辑..3
添加<*FunOrg*><*Unit*>节点..4
添加<*FunStl*><*Unit*>节点..5
添加<*FunLad*><*Unit*>节点..5
添加<*ConvertStl*><*Unit*>节点..6
添加<*ConvertLad*><*Unit*>节点..7
第三步：验证*ManagerFun.xml*文件..7
第四步：修改PLC固件..9
修改*plc_type.h*文件..10
修改*swap_logic.h*文件..10
创建*swap_logic_dic_comw.c*文件..11
编译*swap_logic_dic_comw.c*文件..12
修改*CompileInfor.xml*文件..12
第五步：调试..13

概述

在 GUTTA PLC 系统中添加自定义指令，是进行 PLC 开发的最常见任务。添加自定义指令存在两部分工作。一部分是对上位机软件配置进行修改：首先应该让用户能够在 GUTTA 编程软件中操作自定义指令。这就需要修改指令文件规范 *ManagerFun.xml*。这个文件的详细信息请参考《UM4003 指令描述文件规范》。二部分是对下位机固件进行修改：在编译型 PLC 系统中，需要修改编译配置文件 *CompileInfor.xml*、修改相关头文件、添加自定义指令的 C 语言具体实现并编译生成目标文件。

具体步骤

这里我们以一个简单的例子，一步一步介绍编译系统中添加自定义指令的过程。假设我们需要添加指令的 CPU 类型为 CPU-EC20 (Compile)。需要添加的指令为**数值比较**指令。指令有两个输入操作数，都是整型数值。指令有三个输出操作数，都是位变量。这三个位变量分别在两个输入操作数为大于、等于、小于的时候置位。

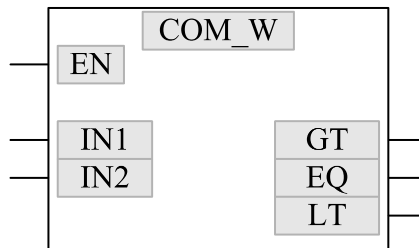
第一步：确定指令格式，绘制图形

在进行任何修改前，首先要明确我们添加指令是什么。由于 GUTTA 支持指令表和梯形图两种模式的编程，故需要定义好两种模式下指令的格式和形状。根据前面提出的指令功能，画出指令草图如下：

指令表指令格式

COMW	IN1,	IN2,	GT,	EQ,	LT

梯形图指令格式



由于不使用组合指令，对应的指令表指令和梯形图指令是一对一的关系。同时这两种操作数都是 5 个，并且他们也是一对一的。需要注意的是，在指令表形式时，指令名是“**COMW**”；在梯形图形式时，指令名是“**COM_W**”；两种名字并不一致。

第二步：修改 *ManagerFun.xml* 文件

找到 *ManagerFun.xml* 文件、打开并编辑

在软件的安装文件夹中，有一个名为 *GuttaLad* 的子文件夹。在 *GuttaLad* 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。每个 CPU 配置下面分别有 *ManagerEnu* 和 *ManagerChs* 这两个子文件夹。这两个文件夹中的文件内容基本一致。只不过对应的语言选项不同。*ManagerEnu* 对应的语言选项为英文，*ManagerChs* 对应的语言选项为中文。我们先修改英文模式下的文件（中文模式下的文件修改基本相同）。

1. 打开 GUTTA Ladder Editor 的安装文件夹。如图：

Name	Size	Type
GuttaLad		File Folder
GuttaLad.exe	2,052 KB	Application
GuttaLad.xml	2 KB	XML Document

2. 打开 *GuttaLad* 文件夹，可以看到若干子文件夹，每个子文件夹代表一种 PLC 配置。

Name	Size	Type
CPU-EC20		File Folder
CPU-EC20 (ARDLABS,CPS-21C)		File Folder
CPU-EC20 (Compile)		File Folder
CPU-EC20 (Cortex-M3)		File Folder

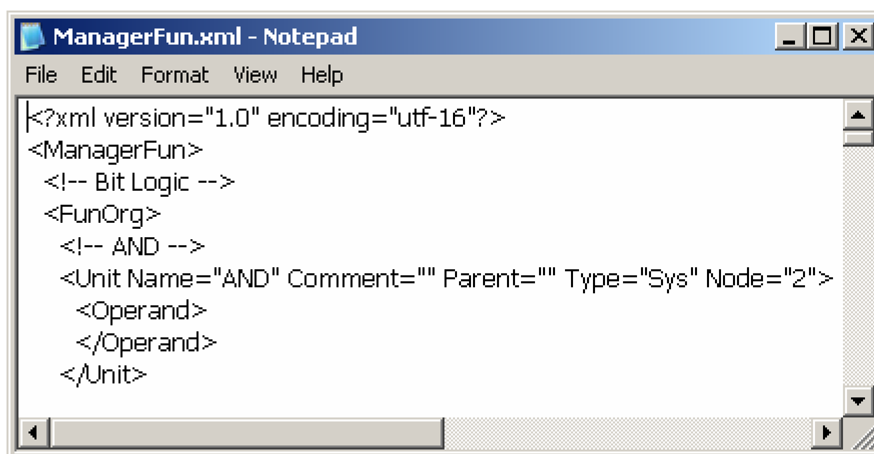
3. 由于我们需要修改的 PLC 类型为 CPU-EC20 (Compile)，故打开上图中 *CPU-EC20 (Compile)*这个文件夹。

Name	Size	Type
CompileFiles		File Folder
ManagerChs		File Folder
ManagerEnu		File Folder
CommunicationDll.dll	596 KB	Application Extension
CompileInfor.xml	17 KB	XML Document
PlcType.xml	2 KB	XML Document
SystemBlockDll.dll	328 KB	Application Extension

4. 对于编译型 PLC，这个文件夹下将有 3 个文件夹和 4 个文件。*PlcType.xml* 是当前 PLC 类型的一个总体描述；*CompileInfor.xml* 是当前 PLC 类型的编译配置；*SystemBlockDll.dll* 负责系统块的编辑；*CommunicationDll.dll* 负责 PLC 程序的上传和下载。我们需要修改的文件就在 *ManagerEnu*（英文模式）中。故打开 *ManagerEnu* 这个文件夹。

Name	Size	Type
ManagerFun.xml	333 KB	XML Document
ManagerVar.xml	9 KB	XML Document

5. 用记事本打开这个文件。



添加<*FunOrg*><*Unit*>节点

参考《UM4003 指令描述文件规范》，写出 COMW 中间指令描述代码如下：

```
<FunOrg>
  <Unit Name="COMW" Comment="" Parent="" Type="Output" Node="1">
    <Operand>
      <FunOperand Name="IN1" Capacity="Wcvp:" Const="Uint" />
      <FunOperand Name="IN2" Capacity="Wcvp:" Const="Uint" />
      <FunOperand Name="GT" Capacity="Tv:" Const="Bit" />
      <FunOperand Name="EQ" Capacity="Tv:" Const="Bit" />
      <FunOperand Name="LT" Capacity="Tv:" Const="Bit" />
    </Operand>
  </Unit>
</FunOrg>
```

1. 第一个节点<FunOrg>表示插入的指令为中间指令。
2. <FunOrg>下的<Unit>节点描述中间指令。
   - Name="COMW" ：描述指令名称，这是 GUTTA 系统识别本指令的依据。这个名称不能被其他中间指令使用。
   - Comment="" ：描述指令的注释，这里省略。
   - Parent="" ：描述指令的子集，这里省略。
   - Type="Output" ：描述指令的类型，由于本指令只有一个能流输入而没有能流输出，这里为输出指令。

- Node="1" ：描述指令的输入能流个数。

3. <Unit>下的<Operand>节点描述描述操作数。
4. <Operand>下的<FunOperand>节点描述中间指令的每个操作数。
    - Name="IN1" ：描述操作数名称。
    - Capacity="Wcvp:" ：描述操作数可以接受的值为字宽度、用法为常数、指针、或变量。
    - Const="Uint" ：描述操作数的数据类型，这里为无符号整数。

由于一共有 5 个<FunOperand>，表示中间指令必须接受 5 个操作数，每个操作数都根据其对应的节点属性来定义。

添加<*FunStl*><*Unit*>节点

参考《UM4003 指令描述文件规范》，写出 COMW 指令表指令描述代码如下：

```
<FunStl>
  <Unit Name="COMW" Comment="" Parent="Additional" Slot="156">
    <Operand>
      <FunOperand Name="IN1" Capacity="Wcvp:" Const="Uint" />
      <FunOperand Name="IN2" Capacity="Wcvp:" Const="Uint" />
      <FunOperand Name="GT" Capacity="Tv:" Const="Bit" />
      <FunOperand Name="EQ" Capacity="Tv:" Const="Bit" />
      <FunOperand Name="LT" Capacity="Tv:" Const="Bit" />
    </Operand>
  </Unit>
</FunStl>
```

5. 第一个节点<FunStl>表示插入的指令为指令表指令。
6. <FunStl>下的<Unit>节点描述指令表指令。
    - Name="COMW" ：描述指令名称。这是 GUTTA 系统识别本指令的依据。这个名称不能被其他指令表指令使用。
    - Comment="" ：描述指令的注释，这里省略。
    - Parent="Additional" ：描述指令的子集名称。
    - Slot="156" ：描述指令的固件代码。这个代码会在后面的固件修改中用到。这个固件代码不能被其他指令表指令使用。
7. <Unit>下的<Operand>节点描述描述操作数。
8. <FunOperand>节点的格式和前面一致。

添加<*FunLad*><*Unit*>节点

参考《UM4003 指令描述文件规范》，写出 COMW 梯形图指令描述代码如下：

```
<FunLad>
  <Unit Name="COMW" Comment="" Parent="Additional" Look="Block" 
Keyword="COM_W">
    <PowerIn>
```

```
      <FunPower Name="EN" />
    </PowerIn>
    <OperandIn>
      <FunOperand Name="IN1" Capacity="Wcvp:" Const="Uint" />
      <FunOperand Name="IN2" Capacity="Wcvp:" Const="Uint" />
    </OperandIn>
    <OperandOut>
      <FunOperand Name="GT" Capacity="Tv:" Const="Bit" />
      <FunOperand Name="EQ" Capacity="Tv:" Const="Bit" />
      <FunOperand Name="LT" Capacity="Tv:" Const="Bit" />
    </OperandOut>
  </Unit>
</FunLad>
```

1. 第一个节点<FunLad>表示插入的指令为指令表指令。
2. <FunStl>下的<Unit>节点描述指令表指令。
   - Name="COMW" ：描述指令名称，GUTTA 系统识别本指令的依据，这个名称不能被其他梯形图指令使用。
   - Comment="" ：描述指令的注释，这里省略。
   - Parent="Additional" ：描述指令的子集名称。
   - Look="Block" ：描述指令的形状，这里为功能块。
   - Keyword="COM_W"：描述指令的名字（显示在功能块正上方）。
3. <Unit>下的<PowerIn>节点描述描述输入能流。
4. <Unit>下的<OperandIn>节点描述描述输入操作数。
5. <Unit>下的<OperandOut>节点描述描述输出操作数。
6. <FunPower>节点只有一个唯一的 Name 属性，描述输入能流的名称。
7. <FunOperand>节点的格式和前面一致。

添加*<ConvertStl><Unit>*节点

参考《UM4003 指令描述文件规范》，写出 COMW 指令表指令到中间指令的规则如下：

```
<ConvertStl>
  <Unit>
    <Left>
      <Item Name="COMW" Key="12345" />
    </Left>
    <Right>
      <Item Name="COMW" Key="12345" />
    </Right>
  </Unit>
</ConvertStl>
```

这个段代码的含义比较简单，就是说指令表指令“COMW”和中间指令“COMW”可以在需要时互相转化，转化时参数一一对应（因为“12345”=“12345”）。

添加*<ConvertLad><Unit>*节点

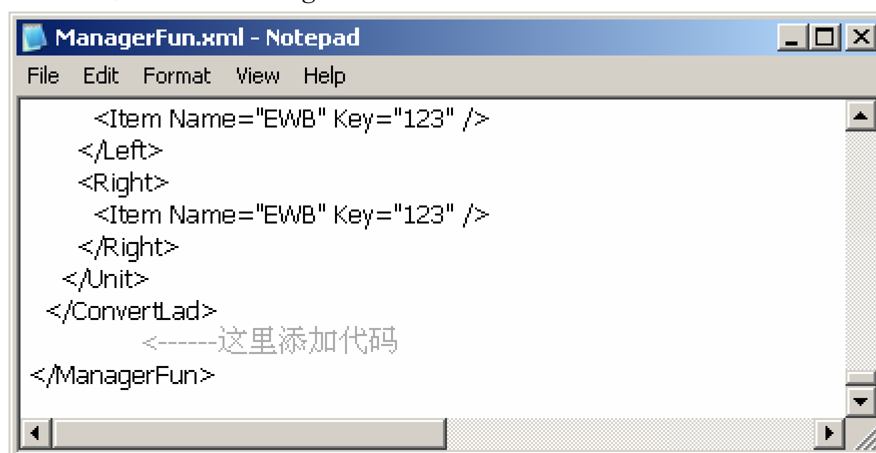
参考《UM4003 指令描述文件规范》，写出 COMW 梯形图指令到中间指令的规则如下：

```
<ConvertLad>
  <Unit>
    <Left>
      <Item Name="COMW" Key="12345" />
    </Left>
    <Right>
      <Item Name="COMW" Key="12345" />
    </Right>
  </Unit>
</ConvertLad>
```

这个段代码的含义比较简单，就是说梯形图指令“COMW”和中间指令“COMW”可以在需要时互相转化，转化时参数一一对应（因为“12345”=“12345”）。

注意到梯形图指令操作数分为标识操作数（本指令没有用到）、输入操作数、输出操作数。在进行一一对应时，梯形图指令按照先标识操作数，后输入操作数，最后输出操作数的顺序匹配。

将上面 5 段代码添加到 *ManagerFun* 节点的最后面：

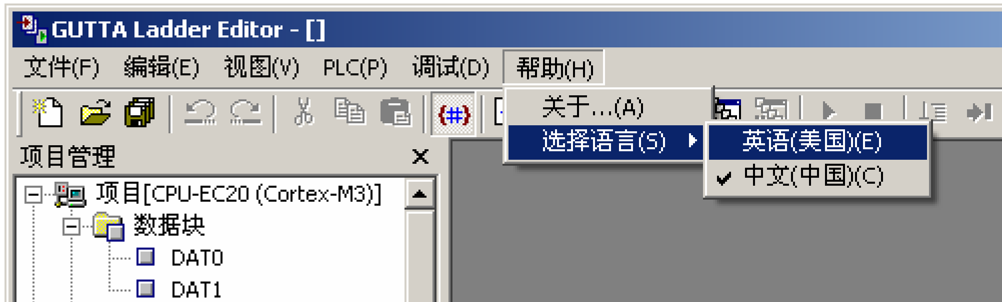


保存文件后，便完成了 *ManagerFun.xml* 文件的修改。

第三步：验证 *ManagerFun.xml* 文件

为了验证以上修改的正确，需要运行 GUTTA Ladder Editor 软件。

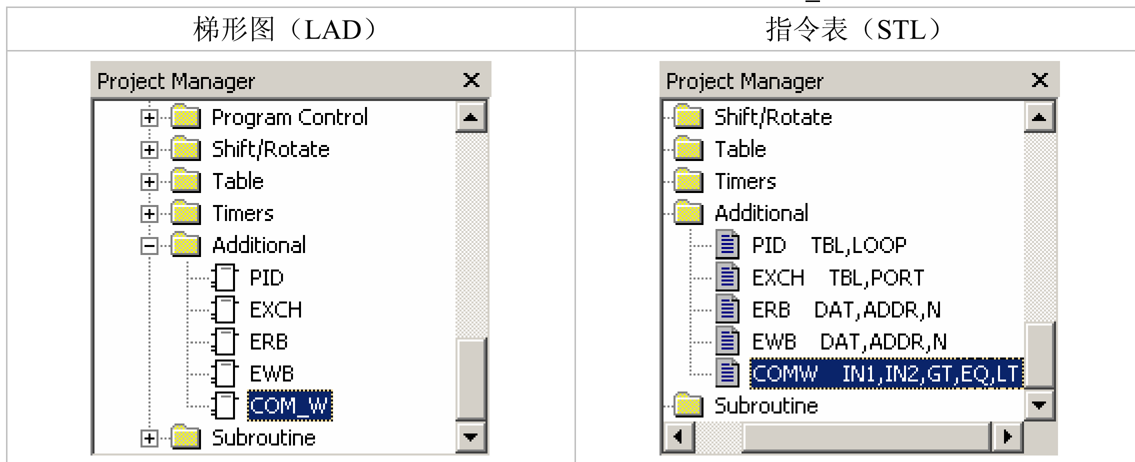
1. 若当前语言设置不是英文，设置语言为英文（因为我们修改的文件为 *ManagerEnu* 文件夹下的 *ManagerFun.xml*），然后重新运行软件。



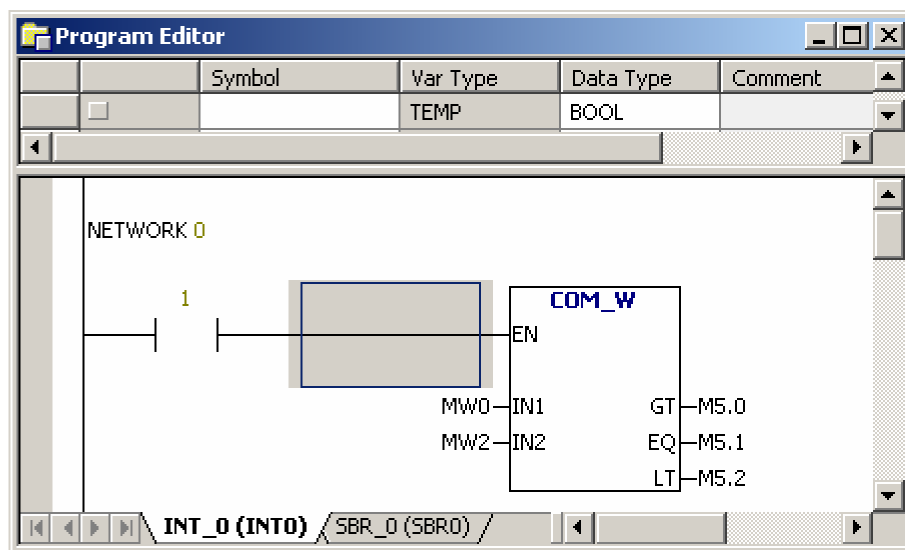
如果在英文模式下，程序能够正常启动，说明修改后的文件在格式上是正确的。

2. 若当前 PLC 类型不是 CPU-EC20 (Compile)，更改 PLC 类型到 CPU-EC20 (Compile)。
3. 观察项目管理窗口中指令下 Additional 子集中是否存在 COM_W 和 COMW 指令。

梯形图（LAD）	指令表（STL）

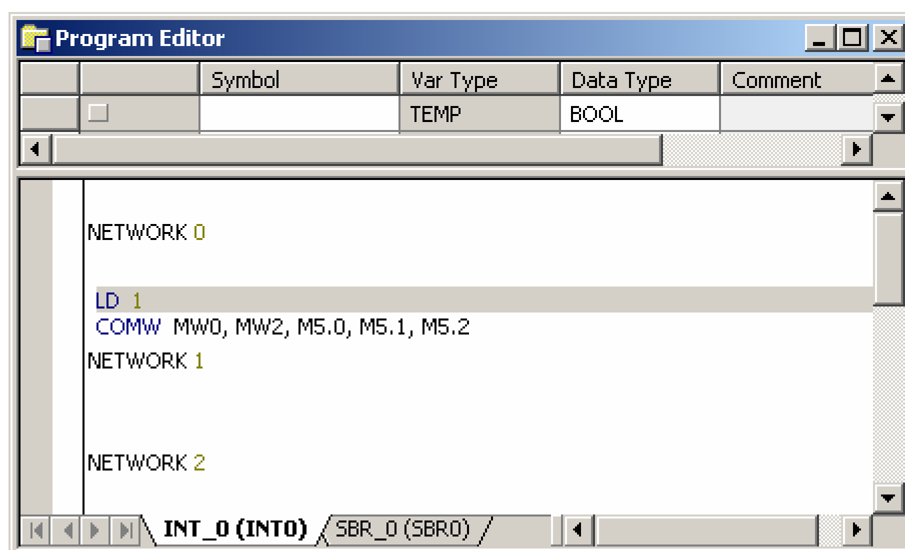


4. 写一个包含该指令的最简梯形图网络，进行一次梯形图到指令表的转化。



5. 写一个包含该指令的最简指令表网络，进行一次指令表到梯形图的转化。

COPYRIGHT © 2008 WWW.VISIBLECONTROL.COM



若两种转换都能够正确执行，说明 *ManagerFun.xml* 文件修改准确无误。由于前面只修改了英文模式下的指令系统，对于中文模式下的指令系统，也要做出相应的修改。这里不再重复讲述。

第四步：修改 PLC 固件

由于编译型 PLC 的固件每次都需要重新编译然后再下载。故修改 PLC 固件，其实就是修改编译所需要的文件。这些文件位于 PLC 配置文件夹下的 *CompileFiles* 子文件夹中：

Name	Size	Type
CompileFiles		File Folder
ManagerChs		File Folder
ManagerEnu		File Folder
CommunicationDll.dll	596 KB	Application Extension
CompileInfor.xml	17 KB	XML Document
PlcType.xml	2 KB	XML Document
SystemBlockDll.dll	328 KB	Application Extension

打开 *CompileFiles* 文件夹：

Name	Size	Type
WinAVR		File Folder
plc_res.h	5 KB	C/C++ Header
plc_type.h	15 KB	C/C++ Header
swap_auto.h	1 KB	C/C++ Header
swap_logic.h	19 KB	C/C++ Header
swap_system.a	76 KB	A File

这个目录下面有 1 个子文件夹和 5 个文件。WinAVR 文件夹用于存放编译器。*plc_res.h*、*plc_type.h*、*swap_logic.h*、*swap_auto.h* 这 4 个文件都是编译时必需的头文件。*swap_system.a* 是一个动态连接库，里面包含了除用户 PLC 逻辑代码以外的 PLC 系统服务。

COPYRIGHT © 2008 WWW.VISIBLECONTROL.COM

修改 *plc_type.h* 文件

```
#endif // FUN_USE_TABLE

#ifdef FUN_USE_TIMERS
                #define FUN_INS_TON          149
                #define FUN_INS_TONR         150
                #define FUN_INS_TOF          151
#endif // FUN_USE_TIMERS

#ifdef FUN_USE_ADDITIONAL
                #define FUN_INS_PID          152
                #define FUN_INS_EXCH         153
                #define FUN_INS_ERB          154
                #define FUN_INS_EWB          155
                #define FUN_INS_COMW         156
#endif // FUN_USE_ADDITIONAL
////////////////////////////////////////////////////////////////////////////
///

// Other:
////////////////////////////////////////////////////////////////////////////
///
#define FUN_INT_SIZE       PLC_TYPE_PROGRAM_BLOCK_PAGE_INT_SIZE
```

在指令表指令中，添加一个 FUN_INS_COMW 的宏定义（表格中黑体部分）。此宏的值等于指令表指令的固件代码。

修改 *swap_logic.h* 文件

```
   #ifdef FUN_INS_EXCH
      void LgcExcuteIns_EXCH(void);
   #endif //FUN_INS_EXCH
   #ifdef FUN_INS_ERB
      void LgcExcuteIns_ERB(void);
   #endif //FUN_INS_ERB
   #ifdef FUN_INS_EWB
      void LgcExcuteIns_EWB(void);
   #endif //FUN_INS_EWB
   #ifdef FUN_INS_COMW
      void LgcExcuteIns_COMW(void);
   #endif //FUN_INS_EWB
/////////////////////////////////////////////////////// swap:
```

添加函数 LgcExcuteIns_COMW()的声明（表格中黑体部分）。

创建 *swap_logic_dic_comw.c* 文件

```
#include "swap_auto.h"

#ifdef FUN_INS_COMW
    void LgcExcuteIns_COMW(void) {
        if (_BT(theRes.itState.itStackData, 0)) {
            // If great
            if (_ADDR_VAL_UINT16(0) > _ADDR_VAL_UINT16(1))
                _ADDR_VAL_BIT_BST(2);
            else
                _ADDR_VAL_BIT_BSF(2);
            // If equal
            if (_ADDR_VAL_UINT16(0) == _ADDR_VAL_UINT16(1))
                _ADDR_VAL_BIT_BST(3);
            else
                _ADDR_VAL_BIT_BSF(3);
            // If less
            if (_ADDR_VAL_UINT16(0) < _ADDR_VAL_UINT16(1))
                _ADDR_VAL_BIT_BST(4);
            else
                _ADDR_VAL_BIT_BSF(4);
            }
    }
#endif //FUN_INS_COMW
```

在文件中加入函数 LgcExcuteIns_COMW()的定义。完成该指令的功能。这里这个指令通过比较两个输入操作数的大小关系，来设置输出参数的值（见该指令的设计要求）。

引用参数时可以使用下列宏：

用_BT(theRes.itState.itStackData, 0)宏来判断当前数据栈栈顶的值。
用_BT(theRes.itState.itStackLogic, 0)宏来判断当前辅助栈栈顶的值。
用_ADDR_VAL_BIT(n)表示第 n 个操作数的位值。
用_ADDR_VAL_BIT_BST(n)将第 n 个操作数的位置位。
用_ADDR_VAL_BIT_BSF(n)将第 n 个操作数的位复位。
用_ADDR_VAL_UINT8(n)表示第 n 个操作数的无符号字节值。
用_ADDR_VAL_UINT16(n)表示第 n 个操作数的无符号字值。
用_ADDR_VAL_UINT32(n)表示第 n 个操作数的无符号双字值。
用_ADDR_VAL_INT8(n)表示第 n 个操作数的有符号字节值。
用_ADDR_VAL_INT16(n)表示第 n 个操作数的有符号字值。
用_ADDR_VAL_INT32(n)表示第 n 个操作数的有符号双字值。

COPYRIGHT © 2008 WWW.VISIBLECONTROL.COM

更多的细节请参考 sfw_logic.h 头文件。

将这个文件保存在 *CompileFiles* 文件夹中，文件名为 *swap_logic_dic_comw.c*。

编译 *swap_logic_dic_comw.c* 文件

在 DOS 窗口中，进入 *CompileFiles* 文件夹，运行编译命令：

```
...\CompileFiles>WinAVR\bin\avr-gcc.exe -c -mmcu=atmega64 -I. -DF_CPU=11059200UL -Os -funsigned-char -funsigned-bitfields -fpack-struct -fshort-enums -Wall -Wstrict-prototypes -Wundef -std=gnu99 -Wundef swap_logic_dic_comw.c -o swap_logic_dic_comw.o
```

这个命令根据 *swap_logic_dic_comw.c* 文件生成对应的目标文件 *swap_logic_dic_comw.o*。

修改 *CompileInfor.xml* 文件

```
<CompileInfor>
  <Instruction>
    <Item Id="0" Name="LgcExcuteIns_ALD" />
    <Item Id="1" Name="LgcExcuteIns_OLD" />
    <Item Id="2" Name="LgcExcuteIns_LPS" />
    ...
    <Item Id="153" Name="LgcExcuteIns_EXCH" />
    <Item Id="154" Name="LgcExcuteIns_ERB" />
    <Item Id="155" Name="LgcExcuteIns_EWB" />
    <Item Id="156" Name="LgcExcuteIns_COMW" />
  </Instruction>
  <Command Directory="\CompileFiles\"
          DirectoryExcute="\CompileFiles\WinAVR\bin\"
          OperateFile="\CompileFiles\swap_auto"
          OperateFileSrc="\CompileFiles\swap_auto.c"
          OperateFileObj="\CompileFiles\swap_auto.o"
          OperateFileHex="\CompileFiles\swap_auto.hex">
    <Enter>
      <Step Value="#avr-gcc.exe -c -mmcu=atmega64 -I. -DF_CPU=11059200UL -Os
-funsigned-char -funsigned-bitfields -fpack-struct -fshort-enums -Wall
-Wstrict-prototypes -Wundef -std=gnu99 -Wundef swap_auto.c -o swap_auto.o" />
      <Step Value="#avr-gcc.exe -mmcu=atmega64 -I. -DF_CPU=11059200UL -Os
-funsigned-char -funsigned-bitfields -fpack-struct -fshort-enums -Wall
-Wstrict-prototypes -Wundef -std=gnu99 -Wundef -L. swap_system.a swap_auto.o
swap_logic_dic_comw.o --output swap_auto.elf" />
      <Step Value="#avr-objcopy.exe -O ihex -R .eeprom swap_auto.elf
swap_auto.hex" />
    </Enter>
```

```
    <Leave>
    </Leave>
  </Command>
</CompileInfor>
```

在 *CompileInfor.xml* 文件中，加入 COMW 指令名 LgcExcuteIns_COMW（表格中黑体部分）；并且在连接命令中加入 *swap_logic_dic_comw.o* 目标文件（表格中黑体部分）。

第五步：调试

在 GUTTA Ladder Editor 软件中新建一个 CPU-EC20 (Compile)项目，输入下面的 PLC 程序。编译下载到 AVR 试验板后，进入调试模式。通过改变 MW10、MW12 的值观察 Q0.0、Q0.1、Q0.2 的值的变化是否正确。