Using and Creating User Defined Function Blocks in ASIC-200

Use of Functions and Function Blocks

Functions in Structured Text

- Functions take one or more input parameters and return one output.
- They are only usable in structured text.
- They have no persistent data associated with them.

```
Output := function(IN1:=value, IN2:=12.34);
```

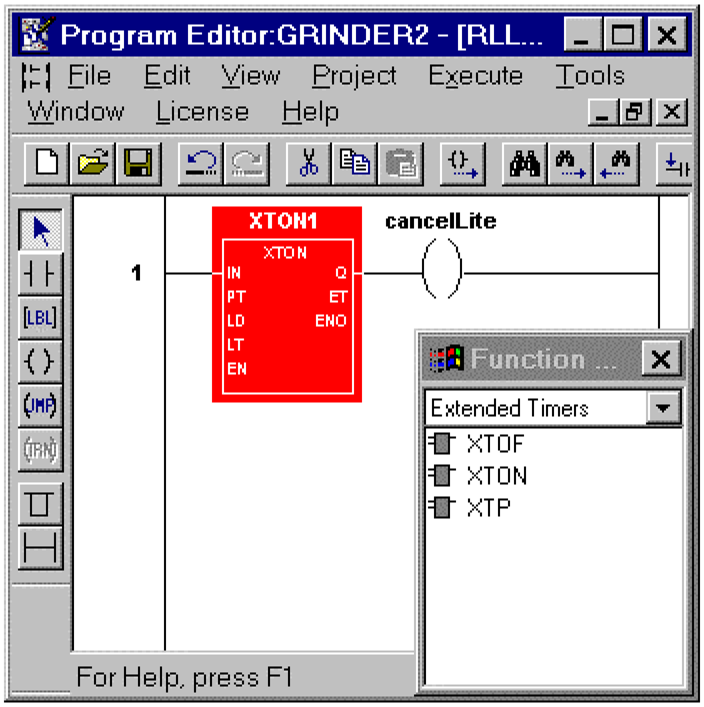
Function Blocks in Structured Text

- Function blocks must be defined like other symbols in the Symbol Manager.
- They can have multiple inputs and outputs.
- The function block designer may define persistent data associated with each block.

```
Fb.IN1:= 12.34;
Fb.IN2:= TRUE;
Fb();
temp:=Fb.OUT1;
temp2:=Fb.OUT2;
```

Function Blocks in Ladder Logic

- Function blocks where the first input and first output are Boolean may be used in ladder logic.
- Ladder logic function blocks appear in the function block palette.
- They are inserted just like the pre-defined function blocks.



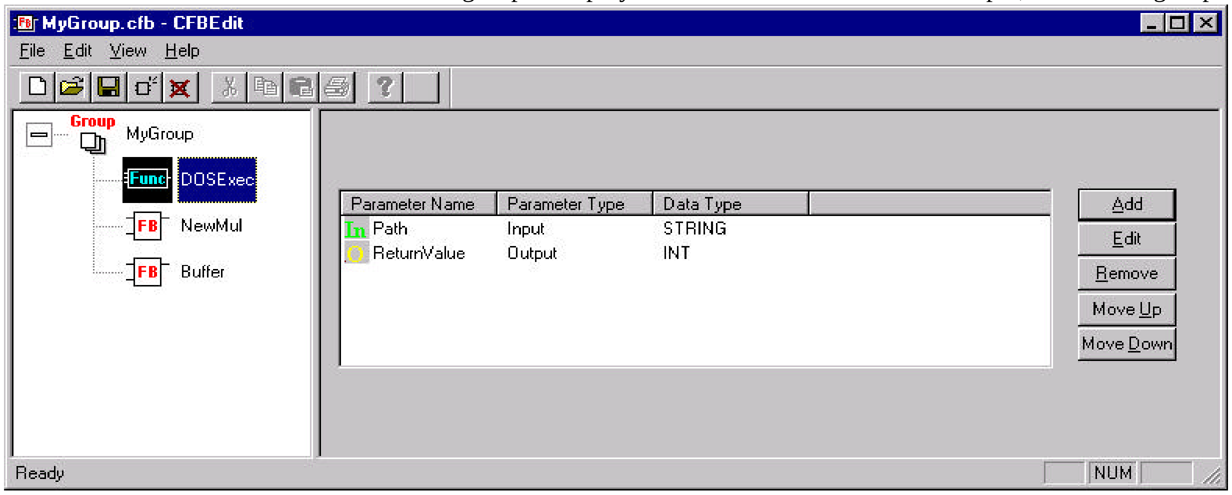
Steps for Creating "C" Functions and Function Blocks

The C Function Block feature allows software developers to create functions and function blocks that can be used in IEC-1131 programs.

All functions and function blocks are associated with a "group." Each group of functions must reside in a user created .DLL file in the \\asic\bin directory. Each group will have its own set of files for connecting to the Runtime Engine and for use with Visual C++.

1. Define your functions and their inputs, outputs, and internal variables. This is best done on paper. There are a few rules to consider while doing this: Functions have no internal data, and no more than one output. Function Blocks can have up to 16 inputs, 16 outputs, and 32 internal variables. NOTE: Internal variables are variables that are associated with the instance of the function block. An example would be a timer function block. A timer function block needs one internal variable – to store the current value. Your C code will have no way of telling the difference between different instances of your function block, so anything that must be stored separately for each instance must be declared as an internal variable.

2. Use the CFBEdit utility to define your functions and function blocks.
   In the CFBEdit window the current group is displayed on the left. In the above example, the current group is



Parameter Name	Parameter Type	Data Type
In Path	Input	STRING
ReturnValue	Output	INT

   called MyGroup. In the tree control on the left the functions and function blocks in this group are displayed. In the window at the right, the inputs, outputs, and internal variables for the selected function or function block are displayed. When you save your group, two files are created in the \bin directory: <groupname>Api.h, which is a header file you will use in your VC project, and <groupname>.cfb, which is used by the Runtime Engine to interface with your DLL. Once this .cfb file is placed in the \bin directory, and the Program Editor is restarted, the group will appear in the RLL editor on the function block palette. NOTE: Only function blocks with a BOOL for the first input and a BOOL for the first output can be used in RLL. Any function can be used in RLL.

3. Create a VC++ project for your function block group. Any DLL that can be called using LoadLibrary and GetProcAddress can be called by the Runtime Engine. If you are going to use MFC, the easiest way to set up your project is as follows: Choose New.. from the File menu, select the project tab, and choose the option "MFC AppWizard (dll)". Create the project with the same name as your group. The default settings should be OK. Once the project has been created, you need to change the settings for the debug version to statically link with the MFC DLLs. Select Settings… from the Project menu. On the left side of the window select Settings for: Win32 Debug. On the right select the C/C++ tab. From the Category list select Code Generation. From the "Use run-time library" list, select Debug Multithreaded. From the Category list select Preprocessor. Remove _AFXDLL from the list of preprocessor definitions.

4. Copy the .h file for your group from the \bin directory to your project directory and add the file to your project.

5. Create functions in your .cpp file using prototypes in the <groupname>Api .h file. You will notice that a function prototype has been created in this file for each function or function block in your group. Each function will take a structure as the only parameter. This structure contains a member for each input, output, or internal variable you specified. In the case of user-defined functions the return type in the C prototype will match the type of the output specified in CFBEdit.

6. Define your function exports in the group.def file for your dll project. This is so that the Runtime Engine will be able to locate the entry points of the DLL. For example, the definition file for a DLL containing functions or function blocks called DOSExec, NewMul, and Buffer would look like this:

```
; MyGroup.def : Declares the module parameters for the DLL.

LIBRARY      "MyGroup"
```

```
DESCRIPTION  'MyGroup Windows Dynamic Link Library'

EXPORTS
    ; Explicit exports can go here

  DOSExec    @1  ;these numbers must be unique
  NewMul     @2
  Buffer     @3
```

7. Build your DLL into the \\asic\bin directory by changing your project settings.

8. To debug your DLL, first test it by writing another program that makes calls into your functions and function blocks. It is best to do this by using LoadLibrary and GetProcAddress to make these calls rather than linking to the DLL, as this is the way the Runtime Engine will be making the calls, but this is not necessary. You can just make a project that makes the calls to the DLL and link to it normally. Testing the DLL in this way is much easier than debugging the DLL while the Runtime Engine is calling into it. Once you are convinced that your DLL works, write a program in the Program Editor that calls one of your functions or function blocks. Start up the Runtime Subsystems and run the program. If further debugging is needed you can debug the DLL as follows: From VC++ select Attach to Process from the Build | Start Debug menu. A list of the running processes will appear. Progscan should be one of the active processes. Select progscan from the list. Once VC is finished attaching to the process open the .cpp file for your .dll. You should now be able to set breakpoints in your DLL.

The Sample Files

Included with this toolkit is an example VC++ project with its corresponding .CFB file. It is in the same directory as this document. To use the example C function blocks in the Program Editor, move the MyGroup.cfb file to the \\asic\bin directory. Then build the project in Visual C++ and move the DLL file to the \bin directory. Three new function blocks will be available in the Program Editor: RunExe, which runs any executable, NewMul, which multiplies two integers, and Buffer, which copies the input to the output delayed by one scan. NewMul and Buffer are very simple, and are a good example of how many function blocks will work. RunExe is more complicated because it has to start a second thread to run the executable. Most function blocks will not need to do this.

Things to keep in mind

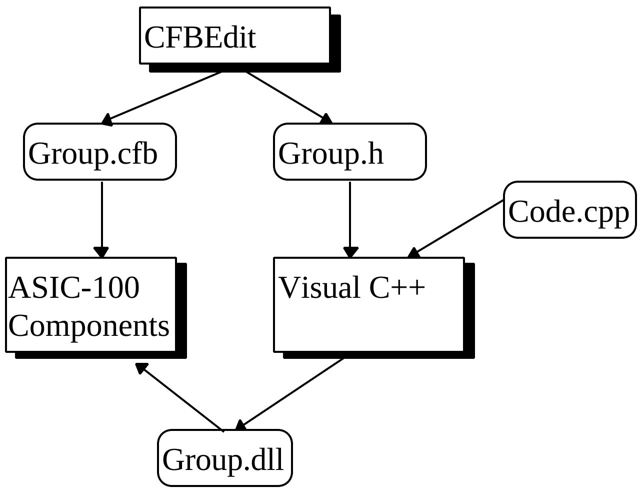
- Your function or function block is executing in the Runtime Engine's logic execution thread, so it is running at the highest available priority under NT. If the execution of your C code takes longer than the current scan rate, the PC will appear to lock up. This happens because the execution thread is using all of the available CPU time, so that there is no time left for anything else to happen. Because of this, it is a good idea to set your scan rate very high when first testing your C code. If your function block needs to do file I/O, communicate with the user, or anything else that could take time, you need to do it in a separate thread. There is an example of how to do this included with this toolkit.
- Variable of type STRING will be passed into functions as type char *. These should be regarded as constant strings and should not be changed.
- Variables of type STRING will be passed into function blocks as a structure:

```
struct ASAPStringType {
      char *m_pchData;     // actual string (null terminated
      int  m_nDataLength;  // does not include terminating 0
      int  m_nAllocLength; // does not include terminating 0
```

```
        };
```

You should NOT reallocate the buffer in this structure. You can change the contents of the buffer up to the allocated size – the size of the allocated buffer is m_nAllocLength. If you change the length of the actual string, you should set m_nDataLength to strlen(new value) before returning.

The diagram below illustrates the use of the tools and the file types created by each tool.


CFBEdit
Group.cfb
Group.h
Code.cpp
ASIC-100 Components
Visual C++
Group.dll