MICROSTAR LABORATORIES™

2265 116th Avenue N.E., Bellevue, WA 98004
Sales & Customer Support: (425) 453-2345
Finance & Administration: (425) 453-9489
Fax: (425) 453-3199
World Wide Web: http://www.mstarlabs.com/

DTD32 Advanced Programming Techniques

This technical note explains some of the additional non-standard programming techniques for the DAPL 32-bit custom command modules which are not described in the manual:

- Assigning a Module Description
- Assigning a Module Copyright label
- Assigning a Module Version
- Changing the default stack size
- Creating global memory regions

Assigning a Module Description and Copyright

If you open the Data Acquisition Processor control program from the Control Panel, click on the Modules tab and select any loaded custom command you will see entries for a module Description, Version, Copyright and Location. By default, all fields except Location will show "not available". To insert a Description and Copyright, add the following lines at the beginning of custom command source code after the #include statements and modify the underlined text following the colon as desired.

```
#pragma comment(exestr,"DaplModuleDescription:...Description goes here...")
#pragma comment(exestr,"DaplModuleCopyright:...Copyright (c) 2002...")
```

Assigning a Version

Assigning a version is not as simple as assigning a Description and Copyright. If the only use will be to visually determine which version is loaded, it is best to place the version information in the Description text. If a complex system of custom command modules will require dependencies to be developed among them, version information may be embedded into the module.

The way this is done is by building the module with the versioning information appended to its name in the form: `name@140@211.dlm` where name is the module name, 140 is the "interface" version, and 211 is the "build" version. This is done by renaming the source code, appending the version data to the given module filename (for example `name.cpp` becomes `name@140@211.cpp`), building the file with the modified name, then renaming the resulting DLM file to the original filename (`name.dlm`). This can be done from a batch file.

Version management can be performed using the functions described in the DTDVER.H file.

Changing the Default Stack Size

The default stack size for a custom command module is 4096 bytes. In general, it is not recommended that the stack size be increased but it is sometimes necessary - for example, if a large number of local variables are used or a recursive function is implemented (in which case a maximum recursion depth should be enforced). Local arrays should be dynamically allocated on the heap using `ralloc()` or `new[]` (`new[]` is required for multidimensional arrays). The C++ syntax to declare a three-dimensional 3 x 12 x 7 integer array on the heap is:

```
int (*aiBuffer)[12][7] = new int[3][12][7];
```

Every custom command has an external reference to the function `ModuleInstall()` which in turn makes a call to `CommandInstall()` for every command in the module. The last parameter of `CommandInstall()` is a reference to a `TCmdProperties` structure which is typically `NULL` but can be used to change certain default command properties including the stack size. It is declared in the header file DTDMOD.H as:

```
typedef struct tag_TCmdProperties
{
  int  iInfoSize;   // Size of this structure. Must be initialized before use.
  long lStackSize;  // stack size in bytes. Multiples of 4096 preferred.
  unsigned long bmFlags;  // bit flags
} TCmdProperties;
```

To change the stack size, a `TCmdProperties` object must be created, `lStackSize` must be set as desired, `iInfoSize` should be set using the `sizeof()` operator. `bmFlags` is reserved for future use and must be set to zero. The reference to this object is then passed to `CommandInstall()`.

Example:

```
int __stdcall ModuleInstall(void *hModule)
{
  TCmdProperties Cmd;
  Cmd.iInfoSize  = sizeof(Cmd);
  Cmd.bmFlags = 0;
  Cmd.lStackSize = 4096*2;  // multiples of 4096 are preferred

  if ( CommandInstall(hModule, COMMAND, ENTRY, &Cmd) )
    return true;
  return false;
}
```

The COMMAND and ENTRY constants are defined at the beginning of a custom command source code if a Microstar custom command example is used as a template. Their values are changed to match the names of the commands being built.

Creating Global Memory Regions

Commands within a single module can share memory among themselves. Typically, this is not desired since there is no control over when each using task is accessing the memory and for basic reader/writer functions it is usually better to use DAPL data structures. There are times, though, when it makes sense to do this.

Simply declaring an item (variable, buffer, object, etc.) at the "global" level (i.e. outside the local scope of either of the individual commands) will create a unique copy of the item for each of the commands. Each item will have the same virtual address but the physical address will be different. In other words, changing the contents of a buffer at address 0x47BF3320 in Command1 will not change the contents of the buffer at address 0x47BF3320 in Command2 because the buffers are in different physical memory.

In order to create a memory region that can be seen by all commands in the module, you need to tell the compiler and linker to create a new internal data region that can be shared. Do this by adding a #pragma line at the start of the memory that will be shared. For example:

```
// ... declarations ...
int giCount;  // NOT global, but each command has its own copy

#pragma data_seg("SharedData")
  int giSharedCount = 0;  // Shared among commands.
  // ... other shared items ...
#pragma data_seg()

// ... rest of code ...
```

In the file modmakem.mak add the linker switch "`/SECTION:SharedData,RWS`"

Note that these changes are specific to the Microsoft compilers.

It is important to initialize the data when it is declared or it will not be shared. To initialize a variable, give it a starting value as shown above. Initialize a static array as (int giArray[100] = {0};). To create global object instances, at least one member must be initialized in the constructor.

Buffers must be created as static arrays. Dynamic buffers cannot be shared since the command that allocates the memory will receive a pointer to a new virtual buffer which will not be accessible to the other commands.