Developing DLLs for Multi-Core Chart Analysis

Beginning with TradeStation 9.1, Chart Analysis windows can be automatically distributed across multiple processor cores, thereby spreading the EasyLanguage calculation load. When the option to enable multiple processor usage is enabled, Chart Analysis windows will automatically be distributed amongst available processors. So if you have four Chart Analysis windows and four processors, there would be one Chart Analysis window on each processor.

When developing DLLs, the distribution of Chart Analysis windows across different processors will need to be taken into consideration. For example:

- When an analysis technique is inserted in Chart Analysis windows residing in different cores, multiple instances of the DLL will be loaded (one for each Chart Analysis core). Therefore, use of global variables in the DLL will apply only to the specific process where the Chart Analysis window is loaded.
- DLLs that are executed in parallel, between multiple Chart Analysis windows, should be thread-safe.

Feedback on some commonly used DLLs:

- **ELCollections** – This DLL has been found not to function correctly when the multi-core feature is enabled.
- **ADE** – This DLL, apparently built on top of ELCollections, has also been found not to function correctly when the multi-core feature is enabled.
- **GlobalVariable** – This DLL has been found to function as expected (using Shared Segement) when the multi-core feature is enabled.

There are numerous ways in which information can be effectively shared across Chart Analysis windows residing in different cores. This document provides several examples, including:

- Global Dictionary Class (existing class)
- Shared Segments
- Memory Mapped Files

Global Dictionary

The Global Dictionary Class allows for the sharing of all EasyLanguage primitive data types across all Tradestation applications that execute EasyLanguage, including Chart Analysis windows residing on different cores. Below are examples of two indicators that function as server and client. These indicators can be inserted in Chart Analysis windows that reside on different processor cores.

Indicator : Share_Value_Server

```
using elsystem.collections;

var: GlobalDictionary gDict(null);

method void AnalysisTechnique_Initialized( elsystem.Object sender,
elsystem.InitializedEventArgs args )
begin
        gDict = GlobalDictionary.Create( true, "SHARING_TEST" );
end;

gDict["close"] = close;
```

Indicator: Share_Value_Client

```
using elsystem.collections;

var: GlobalDictionary gDict(null);

method void AnalysisTechnique_Initialized( elsystem.Object sender,
elsystem.InitializedEventArgs args )
begin
        gDict = GlobalDictionary.Create( true, "SHARING_TEST" );
end;

if ( true = gDict.Contains("close") ) then
begin
        plot1( gDict["close"] astype double );
end;
```

Shared Segment

A Shared Segment uses a regular DLL to export the two functions below:

(.h file)

```
__declspec(dllexport) double __stdcall GetGlobalInfo();
__declspec(dllexport) void __stdcall SetGlobalInfo(double info);
```

Implementation:

(.cpp file)

```
double __stdcall GetGlobalInfo()
{
  return SharedInfoSegment::GetInfo();
}

void __stdcall SetGlobalInfo(double info)
{
  SharedInfoSegment::SetInfo(info);
}
```

SharedInfo is a class defined as follows, with a shared data segment:

(.h file)

```
//-----------------------------------------------------------
// This class will share the member m_info when the class
// gets loaded by different dll instances
//-----------------------------------------------------------
class SharedInfoSegment
{
public:
  SharedInfoSegment() {}
  static void SetInfo(double info) { m_info = info; }
  static double GetInfo() { return m_info; }
protected:
  static double m_info; //value to be shared across processes
};
```

Implementation:

(.cpp file)

```
#pragma data_seg( "SHARED_INFO" )
double SharedInfoSegment::m_info = 0;
#pragma data_seg( )
#pragma comment(linker,"/SECTION:SHARED_INFO,RWS")
```

To test the Shared Segment example, two indicators can be created:

Indicator : _writeInfo

```
external: "myDllName.dll", void, "SetGlobalInfo", double;

SetGlobalInfo( close );
print( "global info updated value = " + NumToStr(close,5) );
```

Indicator : _readInfo

```
external: "myDllName.dll", double, "GetGlobalInfo";

plot1( GetGlobalInfo );
print( "global info received value = " + NumToStr(GetGlobalInfo,5) );
```

The indicators above can be inserted into different Chart Analysis windows that reside on different processor cores.

Memory Mapped File

A Memory Mapped File uses a regular DLL to export the two functions below:

(.h file)

```
__declspec(dllexport) double __stdcall GetGlobalInfo();
__declspec(dllexport) void __stdcall SetGlobalInfo(double info);
```

Implementation:

(.cpp file)

```
//global variable to create one instance of the class
SharedInfoFileMapping gSharedInfoFileMapping("Global\\CLOSE");

double __stdcall GetGlobalInfo()
{
  return gSharedInfoFileMapping.GetInfo();
}

void __stdcall SetGlobalInfo(double info)
{
  gSharedInfoFileMapping.SetInfo(info);
}

//-----------------------------------------------------
// This class will share the instance member m_info
// across processes and also provides synchronization
// functionality trough a global mutex
//-----------------------------------------------------
(.h file)
class SharedInfoFileMapping
{
public:
  SharedInfoFileMapping(char* name) :
    m_hFileMapping(NULL),
    m_pInfo(NULL),
    m_mutex(FALSE, "Global\\SharedInfoFileMappingMutex", NULL)
  {
    Acquire(name);
  }
  ~SharedInfoFileMapping()
  {
    Release();
  }
  void Acquire( char* name );
  void Release();
  void SetInfo(double info);
  double GetInfo();
```

```
protected:
  HANDLE m_hFileMapping;
  CMutex m_mutex;
  double* m_pInfo; //value to be shared across processes
};

(.cpp file)
void SharedInfoFileMapping::Acquire( char* name )
{
  m_hFileMapping = CreateFileMapping (INVALID_HANDLE_VALUE, NULL,
    PAGE_READWRITE, 0, sizeof(double), name );

  if (NULL != m_hFileMapping)
  {
    m_pInfo = (double*) MapViewOfFile( m_hFileMapping,
      FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

    if (m_pInfo == NULL)
    {
      CloseHandle(m_hFileMapping);
    }
  }
}

void SharedInfoFileMapping::Release()
{
  if (m_pInfo)
    UnmapViewOfFile(m_pInfo);

  if (m_hFileMapping)
    CloseHandle(m_hFileMapping);
}

void SharedInfoFileMapping::SetInfo(double info)
{
  if ( NULL != m_pInfo )
  {
    CSingleLock lock(&m_mutex, TRUE);
    *m_pInfo = info;
  }
}

double SharedInfoFileMapping::GetInfo()
{
  if ( NULL != m_pInfo )
  {
    CSingleLock lock(&m_mutex, TRUE);
```

```
    return *m_pInfo;
  }
  else
    return 0.0;
}
```

To test the Memory Mapped File example, two indicators can be created:

Indicator : _writeInfo

```
external: "myDllName.dll", void, "SetGlobalInfo", double;

SetGlobalInfo( close );
print( "global info updated value = " + NumToStr(close,5) );
```

Indicator : _readInfo

```
external: "myDllName.dll", double, "GetGlobalInfo";

plot1( GetGlobalInfo );
print( "global info received value = " + NumToStr(GetGlobalInfo,5) );
```

The indicators above can be inserted into different Chart Analysis windows that reside on different processor cores.