Application Report
SPRA659 - April 2000

Introduction to Code Composer Studio Plugins

Justin B. Helmig *Technical Staff – Wireless Software Applications*

ABSTRACT

A Code Composer Studio plugin allows for the extension or customization of the development and debug environment. The plugin can be a standalone Windows application or an ActiveX control. This application note describes the process of writing a plugin and includes the source for a simple plugin.

Contents

1 Requirements:

- Windows 95, 98, or NT
- Code Composer Studio for Microsoft Windows
- Code Composer Studio Software Development Kit (SDK)
- Microsoft Visual C++ 6.0 or greater

2 Prerequisites:

- Good knowledge of C
- Working knowledge of C++
- Basic knowledge of MFC (for ActiveX control)
- Basic knowledge of Microsoft Developers Studio (Visual C++ IDE)

3 Introduction

Interfacing to Code Composer Studio involves using and deriving one or more of the classes provided in the SDK library, CC_Automation.lib. You must include either the CC_Automation.lib (Release) or CC_AutomationD.lib (Debug) file in the plugin's project.

The main automation object, CCApplication, allows you to access various automation servers, including CCDspUser (debug server), CCProject (project server), CCSourceFileDocuments (edit server), and others. The interface classes are described in the hyperlinked document "CCStudio API Reference" which is included in the SDK library.

For the purposes of this application note, a CCDspUser class is derived to allow access to the debug functionality of Code Composer Studio.

4 Console Application Plugin

Implementing a console application controlling accesses to the debug functionalities of Code Composer Studio involves instantianting a CCDspUser class object. The following steps outline the implementation of the Console Application Plugin:

Create a standard Windows console application project from within Microsoft Developers Studio. In this project include either CC_Automation.lib for Release versions or CC_AutomationD.lib for debug versions.

The first step to interfacing Code Composer Studio is to initialize COM from within the plugin application. Lines 130 through 134 in the sample code listing perform this initialization.

The next step is to create the CCApplication object and select the desired target. Lines 137 through 140 illustrate creating the application.

The top-level control object is the CCDspUser class. The plugin will generally derive from this class as it provides several virtual methods that are called to indicate that certain events have completed. In the example, the plugin's derived class is called CCMyDspUser.

The third step in creating a plugin is to create an instance of this user object. Lines 148 and 149 show the creation of the user object and initializing it to the first board and first task from the Code Composer Studio Setup.

The rest of the plugin example code shows how to develop basic debugging functionality, including: loading a program, running, stepping, setting breakpoints, restarting, writing memory, and reading memory. Notice that several of the requests issued to Code Composer Studio will call a method in the derived CCMyDspUser class. For example when the CCMyDspUser::RequestFileLoad is called, Code Composer Studio will load the file. Upon completion of the file load, Code Composer Studio will call the virtual method CCMyDspUser::OnFileLoaded. Until this method is called, the plugin must be cycling in a Windows message loop. Since the example console application is not event driven, the Windows message loop must be manually implemented as illustrated in lines 177 through 186.

Implementing the functions the plugin will support requires following the set of steps below:

1. Find the desired operation in the API Reference
2. Implement the call. This may involve creating instances of helper classes.
3. Determine if the desired API calls an overridible method in the user class.
   a. If so, override the method in the derived user class
   b. Implement a windows messaging loop while waiting for the Code Composer API to respond
4. Test

5 ActiveX Plugins

ActiveX controls can also be used to interface Code Composer Studio. ActiveX plugins run in process with Code Composer Studio which means that access times to perform Code Composer Studio functions are quicker.

The Code Composer Studio SDK includes a project wizard to create ActiveX controls. The name of the wizard is CCActXWiz.awx.

Creating an ActiveX plugin involves the same processes as creating a stand alone application with a couple of notable exceptions:

- The Code Composer Studio Plugin Project Wizard includes all necessary libraries.
- The user interface is a standards Windows user interface using event driven programming.
- Since the event-processing paradigm is used, the Windows message-processing loop does not have to manually be implemented. The application framework will provide the Windows message-processing loop automatically when the project is created.
- An ActiveX control runs in the same process and thread as Code Composer Studio's user interface thread. Any operations that take a long time will disallow interaction with Code Composer until they complete. The ActiveX component can start a separate thread to do any time-consuming processing.

Initialization and the Code Composer Studio interface classes are the same as in the console application.

Appendix A Code Composer Studio Console Plugin Example

The following code listing implements a Code Composer Studio console application plugin. The example plugin provides some of the primitives used to debug a program. The example was designed for code simplicity and is not necessarily the safest or most elegant way to accomplish the particular operations.

CCSconsole.h

```
 1: #ifndef CCSCONSOLE_H
 2: #define CCSCONSOLE_H
 3:
 4: // Function prototypes
 5:
 6: // Load a program
 7: void LoadProgram( void );
 8:
 9: // Run current program
10: void RunProgram( void );
11:
12: // Single step current program
13: void StepProgram( void );
14:
15: // Resart current program
16: void Restart( void );
17:
18: // Set a memory location
19: void SetMemory( void );
20:
21: // Read a memory location
22: void ReadMemory( void );
23:
24: // Set a breakpoint
25: void SetBreakpoint( void );
26:
27: // Display user menu and get input
28: void RunMenu( void );
29:
30: // Structure for a menu entry
31: typedef struct
32: {
33:     char *cpMenuString;  // String to display
34:     void (*pFunction)(); // Function pointer to service function
35: } tMenu;
36:
37: // Main menu to display to user
38: tMenu menu[] =
39: {
40:   ”(1) Load Program”,   LoadProgram,
41:   ”(2) Set Breakpoint”, SetBreakpoint,
42:   ”(3) Run Program”,    RunProgram,
43:   ”(4) Step Into”,      StepProgram,
44:   ”(5) Restart”,        Restart,
45:   ”(6) Set Memory”,     SetMemory,
46:   ”(7) Read Memory”,    ReadMemory,
47:   ”(8) Exit”,           NULL
```

```
 48: };
 49:
 50: #endif // end CCSCONSOLE_H
```

CCMyDspUser.h

```
 51: #ifndef CCMYDSPUSER_H_
 52: #define CCMYDSPUSER_H_
 53:
 54: // Includes
 55: #include <afxdisp.h>
 56: #include "CCAutomation.h"
 57:
 58: // Class CCMyDspUser inherits from CCDspUser
 59: class CCMyDspUser : public CCDspUser
 60: {
 61: public:
 62:     // Constructor
 63:     CCMyDspUser(const CCDspUser& dspUser) : CCDspUser( dspUser )
 64:     {
 65:         // Initialize flags
 66:         bLoadFlag = true; bHaltFlag = true; bReadFlag = true; bWriteFlag = true;
 67:     }
 68:
 69:     // Overloaded API methods
 70:     virtual void RequestFileLoad(const CString& fullFilePathName , bool symbolsOnly = false) ;
 71:     virtual void Run(CString* condition = NULL , long nIterations = 0 );
 72:     virtual void StepInto(long nIterations = 0 ) ;
 73:     virtual void RequestMemoryRead(CCDspMemory& mem , bool broadcast = false );
 74:     virtual void RequestMemoryWrite(CCDspMemory& mem , bool broadcast = true );
 75:
 76:     // Callback methods
 77:
 78:     // Called when a file is loaded
 79:     virtual void OnFileLoaded( FileLoadError errorCode, bool bSymbolsOnly );
 80:
 81:     // Called when program stops execution
 82:     virtual void OnHalt(const CCDspAddr& currentPC ) ;
 83:
 84:     // Accessor function for the done flag
 85:     bool GetLoadFlag();
 86:
 87:     // Accessor function for the done flag
 88:     bool GetHaltFlag();
 89:
 90:     // Accessor function for the done flag
 91:     bool GetReadFlag();
 92:
 93:     // Accessor function for the done flag
 94:     bool GetWriteFlag();
 95:
 96:     // Called on completed request
 97:     virtual void OnRequestComplete(CCDspMemory& mem , bool requestAborted ) ;
 98:
 99: protected:
100:     bool bLoadFlag;  // current loading
101:     bool bHaltFlag;  // current running state
102:     bool bReadFlag;  // reading memory
103:     bool bWriteFlag; // writing memory
104: };
105:
106: #endif // CCMYDSPUSER_H
```

CCSconsole.cpp

```
107: // CCSconsole.cpp : Defines the entry point for the console application.
108:
109: // Includes
110: #include <stdio.h>
111: #include <afxdisp.h>
112:
113: #include ”stdafx.h”
114: #include ”CCAutomation.h”
115: #include ”CCMyDspUser.h”
116: #include ”CCSconsole.h”
117:
118: // Globals
119: void WaitForMessage( void );
120:
121: CCMyDspUser *pDspUser; // This is the main user object pointer
122:
123: // Main function
124: int main(int argc, char* argv[])
125: {
126:
127:     CCApplication app; // Application
128:
129:     // Initialize COM
130:     if( FAILED(CoInitialize(NULL)) )
131:     {
132:         printf(”FAILED: COM initialization\n”);
133:         return –1;
134:     }
135:
136:     // Startup Code Composer
137:     app = CCApplication::CreateApp();
138:
139:     // Make application visible
140:     app.SetVisible(TRUE);
141:
142:     // Create CCMyDspUser class
143:     // GetDspBoards()  – Gets board types
144:     // GetItem(0)      – Gets first board in list
145:     // GetDspTasks()   – Get tasks associated with board
146:     // GetItem(0)      – Get first task
147:     // CreateDspUser() – Create user associated with task
148:     pDspUser = new CCMyDspUser(
149:         app.GetDspBoards().GetItem(0).GetDspTasks().GetItem(0).CreateDspUser());
150:
151:     // If we have an argument, load it.
152:     if( argc >= 2 )
153:     {
154:          // Load up the specified program (2nd argument)
155:          pDspUser–>RequestFileLoad( argv[1] );
156:
157:          // Wait for load to complete
158:          while( pDspUser–>GetLoadFlag( ) == false )
159:          {
160:              // Wait for the load to complete
161:              WaitForMessage();
```

```
162:          }
163:
164:     }
165:
166:     // Run the menu
167:     RunMenu();
168:
169:     // Clean up COM
170:     CoUninitialize();
171:
172:     return 0;
173:
174: } // endof main()
175:
176: // Wait for a windows message. Return when received
177: void WaitForMessage( void )
178: {
179:     MSG msg; // Windows message
180:
181:     // Dispatch Windows Messages
182:     while( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ))
183:     {
184:         TranslateMessage( &msg );
185:         DispatchMessage( &msg );
186:     }
187:
188: } // endof WaitForMessage()
189:
190:
191: // Prints out main menu and gets user input
192: void RunMenu( void )
193: {
194:     tMenu *pMenu;     // For iterating menu
195:     char   cInput[2]; // Input
196:     int    iItem;     // Integer item entered by user
197:
198:     // Menu loop
199:     while(1)
200:     {
201:
202:         // Reset menu pointer
203:         pMenu = menu;
204:
205:         printf(”\n”);
206:
207:         // Print out the menu
208:         do
209:         {
210:             printf(”%s\n”, pMenu–>cpMenuString);
211:
212:         } while( pMenu++–>pFunction != NULL );
213:
214:         printf(”> ”);
215:
216:         // Get user input
217:         cInput[0] = getchar();
218:         cInput[1] = getchar();
219:         iItem = atoi(cInput) – 1;
```

```
220:
221:         // Validate user input
222:         if( (iItem >= 0) && (iItem < ( sizeof( menu ) / 8 )))
223:         {
224:             // Call function
225:             if( menu[iItem].pFunction != NULL )
226:             {
227:                 menu[iItem].pFunction();
228:             }
229:             else
230:             {
231:                 break;
232:             }
233:         }
234:     }
235:
236:     return;
237:
238: } // endof RunMenu()
239:
240: // Load a program into CCS
241: void LoadProgram( void )
242: {
243:     char cFileName[50]; // Alloc storage for user input on stack
244:
245:     // Get a filename. Don't exceed 50 characters!!
246:     printf("Enter filename to Load> ");
247:     scanf("%s", cFileName);
248:
249:     // Load file
250:     pDspUser->RequestFileLoad( cFileName );
251:
252:     // Wait until load is complete. CCS will indicate it
253:     // is finished by calling the overridden OnFileLoaded
254:     // method. We can poll the loaded flag to see if we are
255:     // done.
256:     while( pDspUser->GetLoadFlag( ) == false )
257:     {
258:         // Wait for the load to complete
259:         WaitForMessage();
260:     }
261:
262: } // end LoadProgram()
263:
264: // Run the program in CCS
265: void RunProgram( void )
266: {
267:     // Start the program running
268:     pDspUser->Run();
269:
270:     // Wait for the program to halt. CCS will indicate the
271:     // program halted (stopped execution) by calling the
272:     // overridden OnHalt() method
273:     while( pDspUser->GetHaltFlag() == false)
274:     {
275:         WaitForMessage(  );
276:     }
277:
```

```
278:     return;
279:
280: } // endof RunProgram()
281:
282: // Single Step in CCS
283: void StepProgram( void )
284: {
285:     // Call the step into method
286:     pDspUser->StepInto();
287:
288:     // Wait for the program to halt
289:     while( pDspUser->GetHaltFlag() == false)
290:     {
291:         WaitForMessage(  );
292:     }
293:
294: } // endof StepProgram()
295:
296: // Reset the PC to the entry point
297: void Restart( void )
298: {
299:     // Reset the PC to the entry point
300:     // No callback method will be called
301:     pDspUser->Restart();
302:
303:     return;
304: } // endof Restart()
305:
306: // Set a memory location
307: void SetMemory( void )
308: {
309:    int iStart; // Start memory address
310:    int iValue; // Value
311:
312:    CCDspValue   val(*pDspUser, enumDSP_LONG); // Value holder
313:    CCDspMemory *pDspMemory;                   // Memory pointer
314:
315:    // Get user input
316:    printf("Enter start memory location> ");
317:    scanf("%x", &iStart );
318:
319:    // Create address and init with user object ptr
320:    CCDspAddr addr(*pDspUser, iStart );
321:
322:    // Create memory object
323:    pDspMemory = new CCDspMemory(*pDspUser, addr, 2, 1) ;
324:
325:    printf("Enter value to write> ");
326:    scanf("%x", &iValue );
327:    printf("\n");
328:
329:    val = (const long) iValue;
330:
331:    // Set the dsp memory to the users value
332:    pDspMemory->SetDspValue( val );
333:
334:    // Request a memory read
335:    pDspUser->RequestMemoryWrite( *pDspMemory );
```

```
336:
337:    // Wait until the read is complete
338:    while( pDspUser->GetWriteFlag() == false )
339:    {
340:        WaitForMessage();
341:    }
342:
343:    delete pDspMemory; // free memory object
344:
345: } // endof SetMemory
346:
347: // Read a memory location
348: void ReadMemory( void )
349: {
350:    int          iStart;     // Start memory address
351:    CCDspMemory *pDspMemory; // DSP memory object
352:
353:    printf("Enter start memory location> ");
354:    scanf("%x", &iStart );
355:
356:    // Create address and init with user object ptr
357:    CCDspAddr addr(*pDspUser, iStart );
358:
359:    // Allocate a memory object to encapsulate the target memory buffer
360:    pDspMemory = new CCDspMemory(*pDspUser, addr, 2, 1) ;
361:
362:    // Request a memory read
363:    pDspUser->RequestMemoryRead( *pDspMemory );
364:
365:    // Wait until the read is complete
366:    while( pDspUser->GetReadFlag() == false )
367:    {
368:        WaitForMessage();
369:    }
370:
371:    delete pDspMemory; // free memory object
372:
373: } // end ReadMemory()
374:
375: // Set a breakpoint
376: void SetBreakpoint( void )
377: {
378:    int iStart; // Start address
379:
380:    // Enter break address
381:    printf("Enter break address> ");
382:    scanf("%x", &iStart );
383:
384:    // Create address and init with user ptr
385:    CCDspAddr addr(*pDspUser, iStart );
386:
387:    // Add the break point. There is no callback method
388:    // for setting a breakpoint
389:    pDspUser->AddBreakPoint( addr );
390:
391: } // endof SetBreakpoint()
392: // Overridden CCDspUser class source
393:
```

CCMyDspUser.cpp

```
394: // Includes
395: #include ”CCMyDspUser.h”
396:
397: // Overloaded API methods
398:
399: // Request file load method – set the flag and call the parent
400: void CCMyDspUser::RequestFileLoad(const CString& fullFilePath , bool symbolsOnly)
401: {
402:     // indicate we are starting an app
403:     bLoadFlag = false;
404:
405:     // Call parent method
406:     CCDspUser::RequestFileLoad(fullFilePath, symbolsOnly);
407:
408: } // endof CCMyDspUser::RequestFileLoad()
409:
410: // Run until completion or breakpoint
411: void CCMyDspUser::Run( CString* condition, long nIterations)
412: {
413:     // Indicate we are exiting the halted state
414:     bHaltFlag = false;
415:
416:     // Call parent method
417:     CCDspUser::Run( condition, nIterations );
418:
419: } // endof CCMyDspUser::Run()
420:
421: void CCMyDspUser::StepInto(long nIterations )
422: {
423:     // Indicate we are exiting the halted state
424:     bHaltFlag = false;
425:
426:     // Call parent method
427:     CCDspUser::StepInto( nIterations );
428:
429: } // endof CCMyDspUser::StepInto()
430:
431: void CCMyDspUser::RequestMemoryRead(CCDspMemory& mem , bool broadcast )
432: {
433:     // Indicate we are requesting a read
434:     bReadFlag = false;
435:
436:     // Call parent method
437:     CCDspUser::RequestMemoryRead( mem, broadcast );
438:
439: } // endof CCMyDspUser::RequestMemoryRead()
440:
441: void CCMyDspUser::RequestMemoryWrite(CCDspMemory& mem , bool broadcast  )
442: {
443:     // Indicate we are requesting a write
444:     bWriteFlag = false;
445:
446:     // Call parent method
447:     CCDspUser::RequestMemoryWrite( mem, broadcast );
448:
```

```
449: } // endof CCMyDspUser::RequestMemoryWrite()
450:
451: // Overload the OnFileLoaded method which gets called upon the completion
452: // of a file load in CCS
453: void CCMyDspUser::OnFileLoaded( FileLoadError errorCode, bool bSymbolsOnly )
454: {
455:     printf(”Load Complete\n”);
456:
457:     // Reset flag, we are done with the load
458:     bLoadFlag = true;
459: }
460:
461: void CCMyDspUser::OnHalt(const CCDspAddr& currentPC )
462: {
463:     // We are entering the halted state
464:     bHaltFlag = true;
465: }
466:
467: // Called on completion of a memory request
468: void CCMyDspUser::OnRequestComplete(CCDspMemory& mem , bool requestAborted )
469: {
470:     CCDspValue val(*this, enumDSP_LONG); // Value holder
471:
472:     // If we issued a read
473:     if( bReadFlag == false )
474:     {
475:         // Get the read value
476:         mem.GetDspValue( val );
477:
478:         printf(”Value Read: %x\n”, val.ToLong() );
479:
480:         // Reset the read flag indicating completion
481:         bReadFlag = true;
482:     }
483:     // If we issued a write
484:     else if( bWriteFlag == false )
485:     {
486:         // Reset the write flag indicating completion
487:         bWriteFlag = true;
488:     }
489:
490: } // endof CCMyDspUser::OnRequestComplete()
491:
492: // Return the done flag member
493: bool CCMyDspUser::GetLoadFlag( void )
494: {
495:     return bLoadFlag;
496: }
497:
498: // Return the done flag member
499: bool CCMyDspUser::GetHaltFlag( void )
500: {
501:     return bHaltFlag;
502: }
503:
504: // Return the read flag member
505: bool CCMyDspUser::GetReadFlag( void )
506: {
```

```
507:     return bReadFlag;
508: }
509:
510: // Return the write flag member
511: bool CCMyDspUser::GetWriteFlag( void )
512: {
513:     return bWriteFlag;
514: }
```

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.