CODELETS

by Aaron Spear

What the heck is a codelet you ask? Simply put, a codelet is a script that runs in the EDGE debugger. But much more than that, codelets are a technology in EDGE that allow you to get inside of your target in ways that simply are not possible in other debuggers. I believe that the hallmark of a truly great technology is that it can be used in ways that the designers never envisioned. Codelets are a technology like this: open ended, your imagination is the limit.

Debugger Scripting History

Of course scripting is not new in the world of debuggers. Most modern embedded debuggers have some sort of scripting capabilities, though for most this is limited to a simple linear sequence of debugger commands. Way back in 1986, Microtec changed the embedded landscape when it introduced the first debugger for embedded systems, XRAY. XRAY 1.0 had an innovative feature called "macros." With a macro, you could define a "C"-like function which you could execute anywhere that you evaluated an expression. That meant that for the first time, you could extend the behavior of the debugger. This feature put XRAY head and shoulders above every other debugger for the next 10 years, and if you look at many debuggers today, their scripting support looks remarkably like XRAY macros.

Fast forward in time a bit to 2002 when Mentor Graphics acquired Accelerated Technology. A team was formed to create a next-generation debugger, taking the best features of the XRAY and code|lab debuggers. Macros were at the top of the feature list, but the question was how could we improve on them? The answer is "codelets," the next-generation evolution of the XRAY macro feature. All of the features of XRAY macros are present in codelets, but the syntax and use of them is much closer to standard "C," making the learning curve in their use shorter.

Codelet Features

Codelets are a full blown "C" scripting language that gives you complete visibility and control over your target application, as well as the ability to interact with and control the debugger. Codelets run in the debugger, but in the context of the target application. They are written in ISO "C" with some extensions, and use standard C flow control (if, for, while, etc). In a codelet you can:

- Read/Write target registers
- Read/Write target memory (anything memory mapped)
- Access target variables symbolically, both locals and globals depending on context
- Call target functions
- Call other codelet functions, including debugger "built-in" functions
- Do I/O with any byte stream via "channels:" read/write to a file, socket, serial port, or a special "Channel Viewer" in the debugger GUI

OK, sounds great, but why is this so unique or special? The magic is the places that you can use codelets, and the resulting power they give you. The first thing to understand is that the codelet execution engine is integrated into the EDGE expression evaluator. So that means that codelets can be evaluated any place that an expression can be evaluated:

- As a condition on a breakpoint
- In the watch window (evaluated any time target state changes, e.g. after stepping)
- Manually via a command line command
- At various hooks provided by the EDGE debug engine during debugging

Channels

"Channels" are new feature to EDGE, not present in any of our previous debuggers. Channels are a multi-drop communications abstraction similar to a named pipe that is built into the core of the EDGE debug engine. Channels don't have anything to do with codelets strictly speaking, but channels open up many possibilities when coupled with codelets.

At its most fundamental level, a channel is simply a bi-directional byte stream. However, each channel has a couple of attributes associated to aid in interpretation of the contents.

- **Encoding:** this specifies the lowest level contents of the stream. values are "binary," "utf-8" text, "utf-16" text, and "utf-32" text.
- **Data type:** This field is a text hint that specifies the interpretation of the contents of the byte stream. It is directly analogous to a mime-type, and is formatted in exactly the same way, utilizing many standard mime-types, and adding extensions.

  Examples are "text/plain," "text/html," "audio/mp3," "video/mpeg," etc.

What is really interesting about channels in EDGE are that we have built frameworks for plugging in different channel sources as well as clients that can connect to them. Some examples of a few channel sources you can use are:

- Files/named pipes (i.e. anything in the file system)
- Sockets
- Serial ports
- Codelets (can both create channels and connect to existing channels)
- Debugged process STDIO (e.g. Microtec SysHost or ARM semihosting)

Coupled with that, in the EDGE GUI we have leveraged the Eclipse plug-in infrastructure to enable both ourselves and customers to write specialized "Channel Viewer" plug-ins. These plug-ins are views in the debugger that receive raw data from a channel and then interpret it. In much the same way as a Web browser can recognize different media types and use different plug-ins to render the content, the EDGE channel manager can connect different viewers to different content types.

In EDGE Tools 1.1.x, there is a number of channel viewers included with the debugger. They include a plain text viewer, a strip-chart recorder that displays values over time, a media player plug-in (wav, mp3, mpeg video, bmp, jpg, png), and the graphics command channel viewer. Things will get much more interesting with the introduction of EDGE 1.2 as we will roll out the ability for customers to write Eclipse/ EDGE channel viewer plug-ins as well as some new built-in viewers you can use (including a VT-100 compatible terminal, html viewer, and binary data viewer – i.e. a memory view) among others.

To me, the most exciting thing about this technology is that there is the possibility of seeing content from your target in a relevant domain-specific rendering. How useful is it to look at a JPEG bitmap in a watch window, or see a generated audio stream in the memory window? No, it is high time that debuggers did a better job in displaying rich content, and that is where we have been going with EDGE since the beginning.

Codelets in Action

SmartWatch™

As I mentioned before, there are a number of hooks inside the EDGE debug engine that can call codelets. One place where this is done is in the evaluation of any expression that results in an instance of a variable of a given type. When this is complete, our expression evaluation engine checks to see if a codelet has been registered for that type, and if it has, it calls that codelet to render the contents of that variable to a character buffer. This string is then displayed for you as a quick way to see the contents of the variable.

Say, for example, that during debugging your application, you hover over a variable that is a pointer to a Nucleus PLUS memory pool. The EDGE editor responds by attempting to resolve what you hover over as a variable on the target and display a tool tip for you with its value. For a structure or class, this is typically just the address of the object. In order to see details about the contents you have to put the instance in the watch window, and that can get annoying. (2 mouse clicks * 10 variables you need to look at * 102 iterations of compile/debug * 8 hours = irritability and tendonitis)

Here is the better solution: a SmartWatch codelet is used to render the contents of the structure to a buffer, and the debugger displays this in a tool tip when we hover over the variable:

```
if (status == NU_SUCCESS)
{
    /* Allocate memory for communication queue */
    status = NU_Allocate_Memory(mem_pool, &pointer,
                                    NU_MEMORY_POOL* mem_pool = 0x00007598 GNED),
                                    SYSMEM    8357028 bytes total
                                    8342556 bytes free (99.8% free)
                                    14472 bytes used (0.2% used)
}                                   0 tasks waiting

/* Check to see if previous operation successful */
if (status == NU_SUCCESS)
{
```

Fig. 1 SmartWatch codelet tool tip

Here is the codelet function that was used to do this.

```
nucleus_smartwatch.cdl | plus_demo.c | tc_defs.h | nucleus.h | esal_ar_isr_a.s
133 */
134 void dump__DM_PCB_STRUCT( char* pBuffer, unsigned long address )
135 {
136     //convert from arbitrary address into valid structure pointer in scope
137     struct DM_PCB_STRUCT* pCB = address;
138
139     sprintf(pBuffer,"\n %s    %u bytes total\n %u bytes free (%3.1f%% free)\n %u bytes
140         (char*)pCB->dm_name,
141         (unsigned)pCB->dm_pool_size,
142         (unsigned)pCB->dm_available,
143         ((double)(pCB->dm_available))/((double)(pCB->dm_pool_size))*100.0,
144         (unsigned)(pCB->dm_pool_size-pCB->dm_available),
145         ((double)(pCB->dm_pool_size-pCB->dm_available))/((double)(pCB->dm_pool_size))*1
146         (unsigned)pCB->dm_tasks_waiting);
147 }
148
149
150 //TODO: mailboxes
```

Fig. 2 SmartWatch codelet source

If you define a codelet function with the prototype "void dump__<TYPE NAME>(char* buffer, unsigned long address)," this function will automatically be called to format a string for you. Cool huh? If you are using EDGE Tools to develop Nucleus PLUS applications right now, ask your AE for the nucleus_smartwatch.cdl file and they will email it to you.

You can of course create your own application specific set of SmartWatch codelets, and share them with your team. Hopefully they will buy you lunch as an expression of their gratitude.

Neil's "Codelet Contest"

Near the end of last year, I sent an email to the entire company encouraging all of our internal engineers to explore the potential of what they could do with codelets and channels in EDGE. Our enterprising and spontaneous GM, Neil Henderson, responded by creating an internal contest to see who could write the most creative and useful codelets (He offered prize money too!). The result was some very interesting entries. I am a firm believer that the best way to have something click in your head is to see tangible examples of it really working.

Nucleus PLUS Insight

"Nucleus PLUS insight" was the winner of the contest. It is a set of codelets written by Irfan Ahmad, an engineer in our RTOS and porting group. The codelet is ~6000 lines long and provides:

1) Implementation of all status commands of the Nucleus SHELL product with identical display format as well as some other miscellaneous tasks
2) Graphical display of task suspension lists for all Nucleus PLUS OS objects
3) Graphical display of memory fragmentation status for Nucleus memory pools

In his codelet, Irfan wrote routines that analyzed and iterated the Nucleus PLUS kernel data structures. For output of information, he sent commands to one of the stock channel viewers that we include with EDGE, the "graphical command channel viewer." Figure 3 is a snippet of the codelet that created the picture above showing the creation of the "Legend" box at the top.

continued on next page

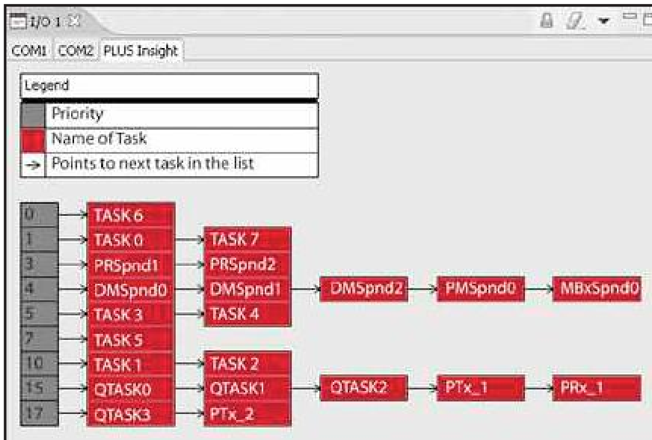

Fig. 3 "PLUS Insight" screenshot

```
gc_setBg(255, 255, 255);
gc_fillRectangle(x, y, boxH+boxW+textW, boxH);
gc_rectangle(x, 10, boxH+boxW+textW, boxH);
@sprintf(buffer, "Legend");
gc_string(&buffer, x+5, y+3);
y += boxH;
```

He utilized a utility codelet library that, for this, took care of the details of sending the graphical commands to the channel.
Below is the implementation of one of these.

```
void    gc_rectangle(int x,
int y, int w, int h)
{
    char     buffer[128];
    int      cnt;

    @sprintf(buffer,
"drawRectangle(%d,%d,%d,%d)",
x, y, w, h);
    cnt = @strlen(buffer);
    @fwrite(chHandle, buffer, 1, cnt);
}
```

Note that we simply use standard ANSI stream I/O routines (e.g. fwrite) to send data to channels from codelets. Nice and simple! Perhaps you are wondering what the story is with the use of the '@' symbol in the routines above? In a codelet, you can call target functions as easily as other codelet functions. In the case that your application provides the implementation of a function such as 'strlen' (good odds), you must specify that you want to call the debuggers version of the function, not the targets.

The '@' sign allows you to do that. We also use the @ sign as an extension to allow us to directly refer to registers, whether they be core registers, or memory-mapped peripheral registers.

SniffLet

Touseef Liaqat, an engineer in our networking technology group, had the great idea to use codelets to provide low-level network packet sniffing for Nucleus NET. This has a number of advantages over traditional network packet sniffers.

1) You don't have to have the packet sniffer in the communication channel. Instead the sniffer is non-intrusively inside one of the embedded nodes. For Ethernet, this isn't critical since you can simply have a packet sniffer running on your workstation, but in the case of protocols such as 802.1X, there may be no way for a host-based sniffer to even know that the network traffic exists.

2) You may be using a strong encryption protocol on the data stream, making an external packet sniffer basically worthless because it cannot decrypt the contents. Using codelets you can plug in extensions to decrypt the contents of the protocol.

Below is a screenshot from EDGE that shows a simple text channel viewer window that Touseef opened, and dumped formatted log info too. The codelet also supports dumping to a text file in PCAP format as well, so you can actually open the file in Ethereal or another packet sniffing type of application for analysis.

Command | Memory 1 | I/O 1 | Build Console
COM1 | ✓ semihosting | SniffLet

```
137.202.156.134   -> 137.202.157.41     TCP   1431  > 7      Flags = 18 Seq = 1806865772   Ack = 4009225992   Win = 64946
137.202.157.41    -> 137.202.156.134    TCP   0     > 0      Flags = 0 Seq = 2311756820   Ack = 0           Win = 202
0:11:43:35:C1:EF  -> FF:FF:FF:FF:FF:FF  ARP   Who has 137.202.157.30 Tell 0:11:43:35:C1:EF
137.202.156.134   -> 137.202.157.41     ICMP  Echo (ping) request
137.202.157.41    -> 137.202.156.134    ICMP  Echo (ping) reply
137.202.156.134   -> 137.202.157.41     TCP   1431  > 7      Flags = 18 Seq = 1806865807   Ack = 4009226027   Win = 64946
127.0.0.1         -> 127.0.0.1          TCP   0     > 0      Flags = 0 Seq = 2311756818   Ack = 0           Win = 202
137.202.157.41    -> 137.202.156.134    TCP   19532 > 19532  Flags = 4c Seq = 1280068684   Ack = 1280068684   Win = 19532
127.0.0.1         -> 127.0.0.1          TCP   48707 > 7      Flags = 10 Seq = 1960808478   Ack = 2240812284   Win = 16095
137.202.156.134   -> 137.202.157.41     TCP   1431  > 7      Flags = 18 Seq = 1806865843   Ack = 4009226063   Win = 64946
0:11:43:35:C1:EF  -> FF:FF:FF:FF:FF:FF  ARP   Who has 137.202.157.152 Tell 0:11:43:35:C1:EF
137.202.156.134   -> 137.202.157.41     ICMP  Echo (ping) request
137.202.157.41    -> 137.202.156.134    ICMP  Echo (ping) reply
127.0.0.1         -> 127.0.0.1          TCP   19532 > 19532  Flags = 4c Seq = 1280068684   Ack = 1280068684   Win = 19532
137.202.157.41    -> 137.202.156.134    TCP   19532 > 19532  Flags = 4c Seq = 1280068684   Ack = 1280068684   Win = 19532
127.0.0.1         -> 127.0.0.1          TCP   7     > 48707  Flags = 10 Seq = 2240812284   Ack = 1960808510   Win = 16016
137.202.156.134   -> 137.202.157.41     TCP   1431  > 7      Flags = 18 Seq = 1806865880   Ack = 4009226100   Win = 64946
```

Fig. 4 "Snifflet" screenshot

Radar Display

John Schneider, one of our AEs, though it would be cool to do an application-specific demo that would mimic something his customers do. He has a number of MilAero customers, and so decided to create a demo system that created and displayed radar data. In Figure 5 (on page 19), John actually opened two EDGE channel viewers in his codelet. One is a standard "strip chart recorder" that displays Radar Scan Load vs. Time. The other is an actual display of the radar data using the EDGE graphical command channel. The radar display also displays different types of plots (radar only, beacon only, correlated, test targets, etc.) in different symbols and colors for easy recognition.

continued on page 19