Interfacing with the CDT debugger, Part 2: Accessing gdb with the Eclipse CDT and MI

How C/C++ Development Tooling uses the C/C++ Debugger Interface to work with the GNU Debugger's Machine Interface

Matthew Scarpino June 24, 2008

The graphical debugging environment provided by Eclipse C/C++ Development Tooling (CDT) is about as good as it gets, displaying breakpoints, watchpoints, variables, registers, disassembly, signals, and memory contents. You can add new capabilities to this environment or access these views to display output from a custom debugger. But first, you need to understand the C/C++ Debugger Interface (CDI) and how it communicates with Eclipse. Part 1 of this "Interfacing with the CDT debugger" series describes the CDI at a high level. Here in Part 2, learn how the CDT talks to the GNU Debugger (gdb). Specifically, learn how the CDT uses the CDI and the Machine Interface (MI) to interface with the gdb.

View more content in this series

The GNU Debugger (gdb) is the most popular open source debugger in use. Originally designed for C, it's been ported to debug code in many languages on a variety of computing systems, from tiny embedded devices to large-scale supercomputers. It's generally used as a command-line executable, but it can be accessed through software using the little-known MI protocol. This article explains how MI works and how the CDT uses MI to communicate with gdb. This concrete example of CDT-debugger interaction should be helpful for anyone wishing to interface a custom C/C++ debugger from CDT.

The Java™ classes discussed here build on the classes and interfaces provided by the CDI, introduced in Part 1 of this "Interfacing with the CDT debugger" series. To remove any confusion, let's be clear about the difference between the CDI and MI:

- The C/C++ Debugger Interface (CDI) was created by Eclipse/CDT developers so CDT can access external debuggers.
- The Machine Interface (MI) was created by gdb developers so external applications can access the gdb.

This may look like a straightforward distinction, but many of the classes I'll present straddle both the CDI and MI, and sometimes it's hard to see where one interface ends and the next begins.

Once you understand how the CDI and MI work together, you'll be better able to link custom debug tools to the CDT, whether they're based on gdb or not

Understanding the GNU Debugger Machine Interface (gdb/MI)

Most people access gdb from a command line, using simple instructions like `run`, `print`, and `info`. This is the *human* interface to gdb. A second method of accessing gdb was developed for interfacing the debugger with software: the Machine Interface (MI). The debugger performs the same tasks as before, but the commands and output responses differ greatly.

An example will make this clear. Let's say you want to debug an application based on the code below.

Listing 1. A simple C application: simple.c

```
int main() {
    int x = 4;
 x += 6;  // x = 10
 x *= 5;  // x = 50
    return (0);
}
```

After you compile the code with `gcc -g -O0 simple.c -o simple`, a regular debug session might look like Listing 2.

Listing 2. A debug session

```
         $ gdb -q simple         (gdb) break main         (gdb) run
1 int main() {
(gdb) step
2 int x = 4;
(gdb) step
3 x += 6;     // x = 10
(gdb) print x
$1 = 4
(gdb) step
4 x *= 5;     // x = 50
(gdb) print x
$2 = 10
(gdb) quit
```

Listing 3 shows how the same gdb session looks using MI commands (shown in bold).

Listing 3. A debug session using MI

```
$ gdb -q -i mi simple
(gdb)
-break-insert-main
^done,bkpt={number="1",type="breakpoint",disp="keep",enabled="y",addr="0x00401075",
func="main",file="simple.c",fullname="/home/mscarpino/simple.c",line="1",times="0"}
(gdb)
-exec-run
^running
(gdb)
*stopped,reason="breakpoint-hit",bkptno="1",thread-id="1",frame={addr="0x00401075",
func="main",args=[],file="simple.c",fullname="/home/mscarpino/simple.c",line="1"}
(gdb)
```

```
-exec-step
^running
(gdb)
*stopped,reason="end-stepping-range",thread-id="1",frame={addr="0x0040107a",
func="main",args=[],file="simple.c",fullname="/home/mscarpino/simple.c",line="2"}
(gdb)
-exec-step
^running
(gdb)
*stopped,reason="end-stepping-range",thread-id="1",frame={addr="0x00401081",
func="main",args=[],file="simple.c",fullname="/home/mscarpino/simple.c",line="3"}
(gdb)
-var-create x_name * x
^done,name="x_name",numchild="0",type="int"
(gdb)
-var-evaluate-expression x_name
^done,value="4"
(gdb)
-exec-step
^running
(gdb)
*stopped,reason="end-stepping-range",thread-id="1",frame={addr="0x00401081",
func="main",args=[],file="simple.c",fullname="/home/mscarpino/simple.c",line="4"}
(gdb)
-var-update x_name
^done,changelist=[{name="x_name",in_scope="true",type_changed="false"}]
(gdb)
-var-evaluate-expression x_name
^done,value="10"
(gdb)
-var-delete x_name
^done,ndeleted="1"
(gdb)
-gdb-exit
```

The `-i mi` flag tells gdb to communicate using the MI protocol, and you can see the difference is significant. The command names have changed dramatically and so has the nature of the output. The first line of the output record is either `^running` or `^done`, followed by result information. This output is called a *result record,* and it can include `^error` and an error message.

In many cases, the MI result record is followed by `(gdb)` and an out-of-band (OOB) record. These records provide additional information about the status of the target or the debugging environment. The `*stopped` message after `-exec-step` is an OOB record that provides information about breakpoints, watchpoints, and why the target has halted or finished. In the previous session, gdb returns `*stopped,reason="end-stepping-range"` after each `-exec-step`, along with the status of the target.

gdb/MI is hard for humans to understand, but it's ideal for communication between software processes. The CDT enables this communication by creating a pseudo-terminal (pty) that sends and receives data. Then, it starts gdb and creates two session objects to manage debug data.

Starting the debugger

As described in Part 1, when the user clicks **Debug**, the CDT accesses an `ICDebugger2` instance and calls on it to create an `ICDISession`. This debugger class must be identified in a plug-in that extends the org.eclipse.cdt.debug.core.CDebugger extension point. Listing 4 shows what this extension looks like in the CDT.

Listing 4. The CDT default debugger extension

```
<extension point="org.eclipse.cdt.debug.core.CDebugger">
   <debugger
         class="org.eclipse.cdt.debug.mi.core.GDBCDIDebugger2"
         cpu="native"
         id="org.eclipse.cdt.debug.mi.core.CDebuggerNew"
         modes="run,core,attach"
         name="gdb Debugger"
         platform="*">
      <buildIdPattern
            pattern="cdt\.managedbuild\.config\.gnu\..*">
      </buildIdPattern>
   </debugger>
</extension>
```

This states that the GDBCDIDebugger2 implements the `createSession()` method that begins the debug process. When the CDT calls this method, it provides the debugger with the launch object containing configuration parameters, the name of the executable to be debugged, and a progress monitor. The GDBCDIDebugger2 uses this information to form a string that starts the gdb executable:

`gdb -q -nw -i` *`mi-version`* `-tty` *`pty-slaveexecutable-name`*.

The GDBCDIDebugger2 creates an `MIProcess` for the running gdb executable, then creates two session objects to manage the rest of the debugging process: `MISession` and `Session`. The `MISession` object manages communication to the gdb, and the `Session` object connects the gdb session to the CDI described in Part 1. The rest of this article discusses these session objects in detail.

The `MISession`

After starting gdb, the first thing the GDBCDIDebugger2 does is create an `MISession` object. This object handles all access to the gdb debugger using three pairs of objects:

- An `OutputStream` to send data to the gdb process and an `InputStream` to receive its response
- An outgoing and incoming `CommandQueue` to hold MI commands
- A `TxThread` that sends commands from the output `CommandQueue` to the `OutputStream` and an `RxThread` that sends receives commands from the `InputStream` and places them in the input `CommandQueue`

An example will demonstrate how these objects work together. If the debug session is conducted remotely, the CDT initiates communication by sending a `remotebaud` command to gdb, followed by the baud rate. To accomplish this, it calls the `MISession`'s `postCommand` method, which adds the `remotebaud` command to the session's outgoing `CommandQueue`. This wakes the `TxThread`, which writes the command to the `OutputStream` connected to the gdb process. It also adds the command to the session's incoming `CommandQueue`.

Meanwhile, the `RxThread` is constantly reading the `InputStream` from the gdb process. When new output is available, the `RxThread` sends it through the `MIParser` to acquire the result record and the OOB record. It then searches through the incoming `CommandQueue` to find the gdb command that

prompted the output. Once the `RxThread` comprehends the gdb's output and the corresponding command, it creates an `MIEvent` used to broadcast the change in the debugger's state.

As data is transferred to and from gdb, the `TxThread` and `RxThread` create and fire `MIEvent`s. For example, if the `TxThread` sends a command changing a breakpoint to gdb, it creates an `MIBreakpointChangedEvent`. If the `RxThread` receives a response from gdb whose result record is `^running`, it creates an `MIRunningEvent`. These events are *not* implementations of the `ICDIEvent` interface described in Part 1. To see how `MIEvent`s and `ICDIEvent`s relate, you need to understand the `Session` object.

`Session`, `Target`, and `EventManager`

After creating the `MISession`, the GDBCDIDebugger2 creates a `Session` object to manage the operation of the CDI. When its constructor is called, the `Session` creates many objects to assist with its management responsibilities. Two objects are particularly important: the `Target`, which manages the CDI model and sends commands to the debugger, and the `EventManager`, which listens for `MIEvent`s created by the debugger.

As Part 1 explains, the `Target` receives debugging commands from the CDT and packages them for the debugger. For example, when you click the **Step Over** button, the CDT finds the current `Target` and calls its `stepOver` method. The `Target` responds by creating an `MIExecNext` command and calling `MISession.postCommand()` to execute the step. The `MISession` adds the command to its outgoing `CommandQueue`, where it's transferred to the debugger in the manner described earlier.

The gdb output, packaged into an `MIEvent`, is received by the session's `EventManager`. When this object is created, it's added as an *Observer* for the running `MISession`. When the `MISession` fires `MIEvent`s, the `EventManager` interprets them and creates corresponding `ICDIEvents`. For example, when the `MISession` fires an `MIRegisterChangedEvent`, the `EventManager` creates a CDI event called `ChangedEvent`. After creating the CDI event, the `EventManager` notifies all interested listeners that a state change has occurred. Many of these listeners are elements in the CDI model, but an important exception is an object called `CDebugTarget`. This is part of another model hierarchy, explained next.

The CDI and the Eclipse debug model

For your debugging plug-in to interface the Eclipse debug views, such as Register View and Variable View, you have to play by Eclipse's rules: You have to use events and model elements taken from the Eclipse debug platform. The root element in the Eclipse debug model is an `IDebugTarget`, and other elements include `IVariable`s, `IExpression`s, and `IThread`s. If these names look familiar, it's because the CDI model hierarchy is structured after the Eclipse debug model hierarchy. But the CDI model and the Eclipse debug model can't talk directly to one another.

For this reason, the CDT contains a set of classes that wrap around CDI classes to provide a bridge between the CDI model and the Eclipse debug model. The `CDebugTarget` is the root of this wrapper-model hierarchy, and it listens for events fired by the CDI `EventManager`. When it receives a new event, the `CDebugTarget` processes a large set of `if` and `switch` statements to determine

how to respond. For example, if the CDI event is an `ICDIResumedEvent`, the `CDebugTarget` executes the code in Listing 5.

Listing 5. Converting CDI events to `DebugEvents`

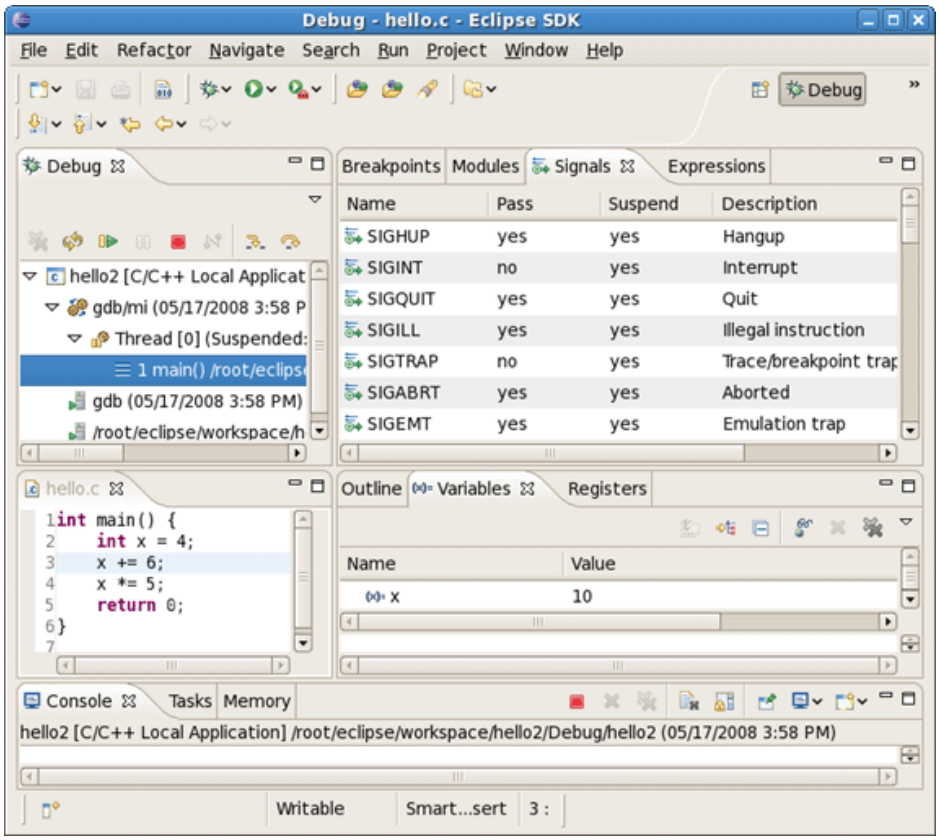
```
switch( event.getType() ) {
 case ICDIResumedEvent.CONTINUE:
  detail = DebugEvent.CLIENT_REQUEST;
  break;
 case ICDIResumedEvent.STEP_INTO:
 case ICDIResumedEvent.STEP_INTO_INSTRUCTION:
  detail = DebugEvent.STEP_INTO;
  break;
 case ICDIResumedEvent.STEP_OVER:
 case ICDIResumedEvent.STEP_OVER_INSTRUCTION:
  detail = DebugEvent.STEP_OVER;
  break;
 case ICDIResumedEvent.STEP_RETURN:
  detail = DebugEvent.STEP_RETURN;
  break;
}
```

The `CDebugTarget` responds to CDI events by creating `DebugEvents`, which are generally related to stepping, breaking, and resuming execution. After creating these events, it accesses the Eclipse `DebugPlugin` and calls its `fireDebugEventSet` method. This notifies all the Eclipse debug listeners that a state change has occurred. That is, any object that adds itself as a `DebugEventListener` receives the `DebugEvent`. This includes the Eclipse debug views, such as the Memory View and the Variables View.

The CDT debug views

The MI-CDI-wrapper-Eclipse communication is useful only if it updates Eclipse's graphical display with proper debug data. Figure 1 shows the CDT debug perspective, and you can see the many views that present the state of the target's execution. Many of the views — Breakpoints, Modules, and Expressions — are provided by Eclipse, but CDT adds three views to the perspective: Executables View, Disassembly View, and Signals.

Figure 1. The CDT debug perspective



These views create and receive debug events in similar ways. This section explains the Signals View. This view, displayed prominently above, lists all the signals the target can receive and shows which can be passed to the process. When the view first appears, the `SignalsViewContentProvider` calls on the `CDebugTarget` to provide a list of signals. This target accesses the CDI target and asks it for the signals in its CDI-model hierarchy. When the array of `ICDISignals` is returned, the `CDebugTarget` updates its own model elements and sends them to the `SignalsViewContentProvider`, which uses them to populate the Signals View.

When you right-click an entry in the Signals View, the **Resume with Signal** context-menu option lets you continue the target's execution and send the selected signal to the process. This option calls on the `SignalsActionDelegate`. When this option is selected, the delegate calls on the CDI target to resume its execution with the `ICDISignal` corresponding to the selected signal. The target creates an MI command for the signal and invokes `MISession.postCommand()`, which sends the command to gdb.

When gdb responds, the process of updating the Signals View takes five steps:

1. The `MISession` analyzes the MI output from gdb and determines whether a signal setting is being changed. If so, it fires an `MISignalChangedEvent`.
2. The CDI `EventManager` listens for the `MISignalChangedEvent` and responds by creating a CDI event: `ChangedEvent`. Then it fires the event and alerts all `ICDIEventListeners`.

3. The `CDebugTarget` receives the event from the `EventManager` and determines whether the `ChangedEvent` relates to a signal change. If so, it calls on its `CSignalManager` to process the CDI event.
4. The `CSignalManager` updates its model elements and fires a `DebugEvent` whose type is given by `DebugEvent.CHANGE`.
5. The `SignalViewEventHandler` receives the `DebugEvent`, checks to make sure it deals with signals, and refreshes the Signals View.

Understanding the involved operation of the Signals View is important for two reasons; It serves as a concrete example of how the different model elements work together, and it shows how you can build similar views that interact with Eclipse, gdb, and the CDI.

Conclusion

Two session objects (`MISession` and `Session`), two targets (`CDebugTarget` and `Target`), and two completely different hierarchies of model elements — the operation of the CDT debugger is so complicated that you may wonder whether any of the developers were related to Rube Goldberg. Still, the code for the CDT debugger was written with modularity in mind, and the better you understand its inner workings the easier it will be to insert your own modules. And remember: The learning curve is steep, but adding new features to the CDT is far easier than building a custom debugging application from scratch.

Downloadable resources

Description	Name	Size
Sample code	os-eclipse-cdt-debug-ex-debugger-plugin.zip	15KB

Related topics

- Visit the Eclipse CDT at Eclipse.org.
- Read the CDT project leader's blog.
- Check out the latest Eclipse technology downloads at IBM alphaWorks.
- Check out the "Recommended Eclipse reading list."
- Download Eclipse Platform and other projects from the Eclipse Foundation.
- Browse all the Eclipse content on developerWorks.
- New to Eclipse? Read the developerWorks article "Get started with Eclipse Platform" to learn its origin and architecture, and how to extend Eclipse with plug-ins.
- Expand your Eclipse skills by checking out IBM developerWorks' Eclipse project resources.