A Framework for Debugging
Olivier, P.A.
3

Case study: A Simulator Framework for Embedded Systems

For this case study we have investigated the use of the TOOLBUS coordination architecture for the development of an embedded system simulator. The simulator developed in this case study is a realistic TOOLBUS application of moderate size (over 25,000 lines of code) that has been used in practice to debug applications for a commercially available embedded system. This case study shows that the TOOLBUS provides an excellent architecture for this kind of application. This simulator is also the first TOOLBUS application in the domain of interactive debuggers, providing us with valuable experience for the work described in the next chapter.

3.1 Introduction

In this chapter, we use the term “embedded system” to identify any computer system for which the primary development tools do not run on the system itself. Typical examples are the computer systems that are built into domestic appliances, cars, and airplanes.

Embedded systems that are sold in large quantities often have excellent development support. State of the art simulators, in-circuit emulators (ICE), etc. are available to aid developers.

But many embedded systems are created for a low volume market, and therefore only a few developers are responsible for the software development of these systems. The overhead of developing tools for these systems using ‘conventional’ methods is just too high. Consequently, these developers still need to rely on print statements and sometimes even memory dumps to debug their software.

It is clear that these developers would benefit enormously from specialized simulators that would boost their productivity and increase the software complexity they can cope with. Programming an embedded system is often a time and money consuming process, and developing and debugging embedded applications can be done much more cost effectively using a simulator instead of using the real hardware all the time.
Dimensions - 130 × 100 × 30mm.
Environmental - -20 degree Celsius to +70 degree Celsius.
- Humidity 40% to 90%.
  Unit is dust and water resistant.
Power supply - 11 Vdc to 29 Vdc.
Microcontroller - Hitachi H8/532 16 bit, running at 10 Mhz.
Boot ROM - 32 Kb.
Flash EPROM - standard 128 Kb, max. 475 Kb.
RAM - standard 32 Kb, max. 512 Kb (Field upgradable).
Retention - 10 year backup with maximum memory.
Communication Ports - 2 RS 232C DTE serial ports.
  1 RS 232C DTE/DCE programmable port.
Connector - 9 Pins sub-D.
Baudrates - 19200, 9600, 4800, 2400, 1200 and many in between.
Format - 1 start bit, 5-8 data bits, even or odd parity bits,
  - 1 or 2 stop bits.

Table 3.1: UPI-10 hardware specifications

This implies that the effort needed to create a sophisticated simulator should be sufficiently small to warrant its creation. This is only possible if we can achieve a significant amount of reuse, both in design and coding efforts.

This is why we deployed the TOOLBUS. This software coordination architecture was designed to control a number of heterogeneous components in a distributed environment. The TOOLBUS enforces formalization of the communication behavior between the components of the system, making the interaction between them very explicit. When exploited wisely, this can lead to a set of loosely coupled tools with a well defined input/output behavior, greatly improving reusability. The TOOLBUS also provides us with the opportunity to implement each component in the most appropriate language.

The ideal simulator provides all the information about the state of the running program the user wants, and no more. To provide this information in a clear and concise way, a good and extensive user interface is a must. The considerable effort that must be invested into the creation of appealing and easy to use user interface components, logically focussed our attention on making these components reusable.
There are a lot of devices on the market that are equipped with a serial port for external communication. Examples include terminals, (radio-)modems, mobile printers, LCD displays, mobile phones, etc. Although the majority of these devices use the RS232 protocol at the lowest level, no consensus exists about which higher level protocol to use. Consequently, most manufacturers choose a protocol best suited for the applications they are interested in. Examples include the vt100 and ANSI escape codes for terminals, and a whole range of protocols for mobile communication, like MOBITEX, MAP 27, and RD LAP Ardis. The UPI-10 (Universal Protocol Interface) was designed as a universal way to interconnect these devices. Figure 3.1 shows a picture of the UPI-10.

The UPI-10 offers three serial ports, a powerful microcontroller, 128Kb Flash ROM and 32Kb RAM. This makes it possible to interconnect three completely different RS232 devices\(^1\) and program the UPI-10 to interface between the different protocols used by these devices.

### 3.2.1 The Software

A piece of hardware like the UPI-10 is useless without a suitable development environment. Because the UPI-10 was meant to be a low cost product, using third party tools was not an option. We developed an assembler, a linker and a back end for \textit{lcc}, a retargetable C compiler

\(^1\)Even more when a serial multiplexer is used.
described in [FH95]. In addition, we implemented a runtime library containing a subset of the ANSI-C routines ([KR88]) and some communication protocols.

The only thing missing was a powerful simulator. The project described in this chapter resulted in such a simulator that is used to debug and test both the development tools as well as the actual programs developed for the UPI-10.

### 3.2.2 The Simulator

Before turning our attention to the architecture of the simulator, we will first give an impression of the functionality it has to offer. Figure 3.2 gives an overview of the user interface components of the simulator.

After starting the simulator, the user is confronted with the main control center, shown in the center of Figure 3.2. The user can load executables and/or symbol tables and start up the other components, which can be divided into three categories:

- Assembler level debugging components.
- Source level debugging components.
- Communication debugging components.

The assembler level debugging components can be used to inspect and interact with the system at the level of the CPU. The components are shown on the left hand side of Figure 3.2:

- *The memory viewer* is used to inspect and change the contents of the memory of the simulated UPI-10.
- *The CPU viewer* enables the user to inspect and change the contents of the CPU registers.
- *The assembler viewer* displays the disassembled memory contents. It allows the user to step through the assembler code and toggle breakpoints.

The source level debugging components handle debugging at the source code level. These components are shown in the middle of Figure 3.2:

- *The source viewer* tracks the current source file and the current point of execution within this file. Breakpoints are highlighted and can be changed.
- The *variable viewer* tracks the contents of variables and lets the user change them.

The communication components simulate the communication between the UPI-10 and any connected RS232 devices. So far, we have implemented two of these components, both of which are shown on the right hand side of Figure 3.2:
Figure 3.2: An overview of the simulator
The communication spy keeps track of all the connections established in the system and shows a nice picture of these connections, together with a list of all the device types present in the system.

The terminal emulator is the only RS232 device simulator we developed so far. It emulates a simple terminal that can be connected to one of the UPI-10 serial ports or to another RS232 device simulator.

### 3.3 Simulator Architecture

Because we aim to create a simulator system that is highly modular and contains a number of reusable components, the basic architecture should be general enough to handle a wide variety of computer systems.

![Simulator Design: Component Overview](image)

Figure 3.3 shows a diagram of the architecture of our simulator framework. It shows the TOOLBUS surrounded by the different tools. Within the TOOLBUS, the processes are depicted. It is beyond the scope of this thesis to discuss the whole system in detail, so we will outline the general architecture and only present the processes SRCVIEW, BREAKS, and EXEC (see Figure 3.3) in more detail. These processes form a small but coherent subsystem and give a good impression of the kind of communication patterns the system is based on. A more extensive presentation of the internals of the system can be found in [Oli96a].
3.3.1 Virtual Machine

Central to the architecture is the notion of a virtual machine. This component has no user interface, but takes care of the actual execution of simulated programs. The virtual machine must simulate all parts of the system that have to run at full speed, so no TOOLBUS communication is needed during the high-speed execution of a program section.\(^2\) To accomplish this, the virtual machine has to keep track of the following information:

- CPU register contents.
- Memory contents.
- The status of any simulated special hardware needed during execution, like switches, LED's, communication hardware etc.
- All breakpoint activity.
- Debug symbols (to map addresses to line numbers etc.).

In addition to the 'bookkeeping' tasks needed to maintain this information, the virtual machine performs the following operations:

- Simulate actual execution of machine language instructions.
- Disassemble the contents of memory on request.

All the other tools, except for the configuration database, provide views on the different types of information maintained by the virtual machine, and enable the user to manipulate this information.

The configuration database is used to maintain sets of options, so every user can configure the system to suit his/her preferences.

3.3.2 The Source Viewer Subsystem

We will now present a coherent part of the simulator's TOOLBUS script that clearly demonstrates the system's organization. This subsystem is formed by the three processes named explicitly in Figure 3.3, and describe the communication between the virtual machine and the source viewer.

The EXEC process handles all communication directly related to the execution of simulated programs. Its tasks are:

- Starting and stopping the execution of the virtual machine.

\(^2\)We use the term 'program section' in this context to indicate a portion of the program to be executed without a need for visual feedback to the user. For instance, execution of the code generated by one line of source code or by an entire routine depending on the command issued by the user.
Retrieving the current execution status of the virtual machine (either running or stopped).

Catching line-number events generated by the virtual machine, and broadcasting them as notes to the rest of the system.

This is expressed in the following TOOLBUS script:

```
process EXEC(Vm : vmtool) is
  let
    Level : term,
    Action : term,
    Status : term,
    Module : str,
    Line : int
  in
    ( % % Handle requests to start execution.
      rec-msg(exec, Level?, Action?) .
      snd-do(Vm, exec(Level, Action))
      +
      % % Handle requests to stop execution.
      rec-msg(exec, user-break) .
      snd-do(Vm, user-break)
      +
      % % Handle status information requests.
      rec-msg(exec, get-exec-status) .
      snd-eval(Vm, get-exec-status) .
      rec-value(Vm, exec-status(Status?)) .
      snd-msg(exec, exec-status(Status))
      +
      % % The virtual machine can inform us that the current % % line number has changed.
      rec-event(Vm, line-number(Module?, Line?)) .
      snd-note(line-number(Module, Line)) .
      snd-ack-event(Vm, line-number(Module, Line))
    ) * delta
endlet
```

The process BREAKS handles all communications related to the status of breakpoints. Its tasks are:

- Setting, clearing, and toggling of breakpoints.

- Retrieving the list of all breakpoints maintained by the virtual machine.
Informing everyone of changes in the list of breakpoints maintained by the virtual machine.

This is expressed in the following TOOLBUS script:

```plaintext
process BREAKS(Vm : vmtool) is
let
   Adr      : int,
   Breaks   : list,
   Module   : str,
   Line     : int
in
   (    % % Individual breakpoint toggling.
       rec-msg(breaks, toggle(Adr)) .
       snd-do(Vm, toggle-break(Adr))

   +    % % Turn of all breakpoints related to one source line.
       rec-msg(breaks, line-off(Module?, Line?)) .
       snd-do(Vm, line-break-off(Module, Line))
   +    % % Turn on all breakpoints related to one source line.
       rec-msg(breaks, line-on(Module?, Line?)) .
       snd-do(Vm, line-break-on(Module, Line))
   +    % % Handle requests for the list of current breakpoints.
       rec-msg(breaks, get-breaks) .
       snd-eval(Vm, get-breaks) .
       rec-value(Vm, breakpoints(Breaks?)) .
       snd-msg(breakpoints, breakpoints(Breaks))
   +    % % When a change has taken place in the list of
       rec-event(Vm, breaks(Breaks?)) .
       snd-note(breaks-changed(Breaks)) .
       snd-ack-event(Vm, breaks(Breaks))
   ) * delta
endlet
```

On the source viewer side, the SRCVIEW process handles all communication, its tasks are:

- The creation of new source-viewer windows.
- Informing the source-viewer of changes in the execution status of the virtual machine.
• Informing the source-viewer of changes in the list of breakpoints maintained by the virtual machine.

• Receiving requests to turn on/off breakpoints and sending them to the BREAKS process.

• Receiving requests to start or stop execution, and sending them to the EXEC process.

This is expressed in the following TOOLBUS script:

process SRCVIEW is
  let
    Src   : srcview,
    Level : term,
    Action: term,
    Module : str,
    Line  : int,
    Breaks : list
  in
    % % Start the source viewer tool.
    execute(srcview, Src?) .
    % % Subscribe to some interesting note types.
    subscribe(line-number(<str>, <int>)) .
    subscribe(breaks-changed(<list>)) .
    subscribe(config-changed) .
    ( % % Handle source-viewer window creation requests
      rec-msg(srcview, create-view) .
      snd-do(Src, create-view)
      +
      % % The program counter has reached a new source line.
      rec-note(line-number(Module?, Line?)) .
      snd-do(Src, line-number(Module, Line))
      +
      % % The breakpoint information has changed.
      rec-note(breaks-changed(Breaks?)) .
      snd-do(Src, breaks-changed(Breaks))
      +
      % % The user wants to set a breakpoint. Propagate the % request to the BREAKS process.
      rec-event(Src, break-line-on(Module?, Line?)) .
      snd-msg(breaks, line-on(Module, Line)) .
      snd-ack-event(Src, break-line-on(Module, Line))
      +
      % % Idem for turning off a breakpoint.

34
The whole interconnection layer of the system consists of these kinds of simple TOOLBUS scripts that describe the communication patterns between components. Studying these well organized TOOLBUS scripts makes it possible to gain a high level understanding of a system without needing to know all implementation details. In cases where more detail is needed, the source code of the different tools can be consulted.

### 3.4 Conclusions

In this section, we will present some of the conclusions that can be drawn from this case study.

#### 3.4.1 Constraints

A system has to satisfy several constraints in order to fit in this simulator framework. When faced with a project that might benefit from this work, these constraints might be a good place to start your evaluation.

**Hardware architecture**  The framework is biased towards what one might call a classic architecture. This means a system containing a single processor\(^3\) that reads its instruction and data streams from a central store. The simulated processor must at least implement a program counter so the other tools can track the execution and a frame pointer if there are any stack based (local) variables.

\(^3\)We do not exclude a multi-processor environment, but this would require some work.
Table 3.2: Simulator dependencies

**Software support**  To be able to make sense of the contents of the simulated memory and register contents, the simulator needs static symbol information. In most cases this information is generated by the linker. Table 3.2 shows the information needed and the tools that depend on it. A dagger (†) in a certain column means that that particular tool uses the feature mentioned in the first column, but still works without it (albeit with some reduced functionality). A ‡ means that a tool depends on a feature and is pretty useless without it.

### 3.4.2 Evaluation

How well did we achieve the reusability and performance targets mentioned in the introduction of this chapter?

**Reusability Targets**  The reusability of the system is best illustrated by some statistics about the size of the software (Table 3.3). We focus on reusability of components when using different CPU’s. Typically, compiler backend, assembler, linker, and virtual machine will then have to be replaced. The architecture of the simulator makes it easy to add new components when other elements change besides the CPU.

Of course, Table 3.3 only provides a very rough estimate, mainly because the implementation languages are so diverse.

An obvious conclusion is that the simulator size is almost as big as the other development tools together, warranting a major investment to reuse parts of it.

---

<table>
<thead>
<tr>
<th>static features</th>
<th>Control Center</th>
<th>CPU Viewer</th>
<th>Memory Viewer</th>
<th>Source Viewer</th>
<th>Variable Viewer</th>
</tr>
</thead>
<tbody>
<tr>
<td>address labels</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
</tr>
<tr>
<td>line number info</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
</tr>
<tr>
<td>function info</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>‡</td>
</tr>
<tr>
<td>variable info</td>
<td></td>
<td></td>
<td></td>
<td>‡</td>
<td></td>
</tr>
<tr>
<td>scope info</td>
<td></td>
<td></td>
<td></td>
<td>‡</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dynamic features</th>
<th>Control Center</th>
<th>CPU Viewer</th>
<th>Memory Viewer</th>
<th>Source Viewer</th>
<th>Variable Viewer</th>
</tr>
</thead>
<tbody>
<tr>
<td>program counter</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cycle counter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>‡</td>
</tr>
<tr>
<td>frame pointer</td>
<td></td>
<td></td>
<td></td>
<td>‡</td>
<td></td>
</tr>
<tr>
<td>cur. scope list</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>†</td>
</tr>
<tr>
<td>breakpoints</td>
<td></td>
<td></td>
<td></td>
<td>†</td>
<td></td>
</tr>
</tbody>
</table>

---

36
Conclusions

3.4

Table 3.3: Some code statistics

In a new simulator, the UI components and the TOOLBUS script are directly reusable. This means an instant reuse of 36 percent of high-level simulator code.

Although the virtual machine is marked as non reusable, this is not strictly the case. This project was centered around the reuse of complete tools. Because the virtual machine is nothing but a very specialized program, traditional techniques aimed at reusability can be applied very successfully.

Performance Targets  Because the actual program execution takes place within in the virtual machine, the simulator performance in this area is not influenced by the general simulator architecture. When running the virtual machine on a 100 Mhz Pentium PC, it outperforms the actual UPI-10 hardware so delays need to be introduced to make the behavior more realistic.

Although the user interface components are all separate tools, requiring inter-process communication on almost every user interaction, the interactive response times are quite good, essentially justifying the choice for the TOOLBUS concept.

3.4.3 TOOLBUS Experience

One of the goals of this project was to gain experience with TOOLBUS programming. Now that we have implemented one of the first medium sized TOOLBUS applications, some preliminary conclusions can be drawn.

- The TOOLBUS is very good in separating components, especially if the traffic between
An interesting separation technique is the complete separation between the user interface and computational components. Because the TOOLBUS communication is very fast compared to a user interacting with a user interface, only a small performance penalty is paid by separating the components.

The one-to-one mapping between tools and programs is very unsatisfactory. The need for interprocess communication for every TOOLBUS/tool interaction seriously affects performance.

The only reason why we did not split up the system even further, for instance by separating the symbol table manager and the actual virtual machine, was a lack of performance. This suggests a general strategy when designing a system around the TOOLBUS.

- First start with an (existing?) design where all components are located in one tool.
- Move all user interface components into separate tools.
- Try to isolate other components that can be turned into tools without seriously affecting the overall performance.

Several studies on micro-kernel operating systems (for instance the Amoeba operating system, [RvRT89]) have shown that it is possible to get a decent performance while using message passing. The possibility to incorporate some tools in the TOOLBUS instead of connecting them using interprocess communication primitives will make a big difference and opens up a wide range of application areas.

3.4.4 Future Research

We definitely need to develop more simulators, in order to refine and extend this framework.

An interesting experiment would be to replace the virtual machine tool with a symbol table manager and a program debugging interface to programs running on the host machine, so the tool can monitor and interact with these programs. This would extend the framework to incorporate debuggers as well as simulators without changing the user interface components. Chapter 4 describes a debugging framework that has been inspired by the work described in this Chapter.

On the TOOLBUS side, a number of interesting fields lay open for investigation:

- How much can we boost performance by moving tools into the TOOLBUS thus eliminating the interprocess communication?
- The dynamic loading of new TOOLBUS scripts seems to be a logical next step, complementing the dynamic creation of processes and the dynamic connection/disconnection of tools, both of which are already present.
• The TOOLBUS script for the simulator grew to considerable size (more than 1,600 lines), but was still manageable. As we gain experience with the TOOLBUS, our scripts will grow and we will need some kind of modularization construct.

• The TOOLBUS uses a string representation of terms to communicate with the tools. This makes it very hard to share common subterms. Experimenting with an architecture independent term or graph representation that preserves sharing would be interesting. In Chapter 6 we show how this kind of sharing can be achieved.