ID 310C: Run-Time Visualization on Renesas MCUs

Micrïµm
Matt Gordon
Sr. Applications Engineer
12 October 2010
Version: 1.2
Matt Gordon

- Sr. Applications Engineer
  - Responsible for demos and example projects
  - Multiple articles and white papers
  - Head of Micrium’s training program

- Previous Experience
  - Software engineer at Micrium
    - Developed device drivers and kernel ports
  - Bachelor’s degree in computer engineering from Georgia Tech
In the session 110C, Renesas Next Generation Microcontroller and Microprocessor Technology Roadmap, Ritesh Tyagi introduces this high level image of where the Renesas Products fit. The big picture.
This is where our session, ‘ID 310C: Run-Time Visualization on Renesas MCUs’ is focused within the ‘Big picture of Renesas Products’
Here are the MCU and MPU Product Lines, I am not going to cover any specific information on these families, but rather I want to show you where this session is focused <CLICK>” NOTE For Reviewer, the below notes are from the 110C presentation so you can better understand this slide

**Notes for Devcon Positioning Slide:** There’s a lot of vital information on this slide, which spotlights the Renesas MCU/MPU product lines recommended for new designs. Perhaps the best way to discuss this material is to cover it from a very high level.

Since the merger, we have scrutinized the needs of our global markets, reassessed our strengths, and implemented a business strategy focusing on supporting the ‘ubiquitous computing’ paradigm. This insightful concept — often abbreviated as ‘ubicomp’, and sometimes termed ‘pervasive computing’ or ‘ambient intelligence’ — was introduced by Mark Weiser of Xerox in 1988.

Ubiquitous computing refers to a new genre of computing, a worldwide electronic environment in which computer-controlled products completely permeate the life of end users around the globe. Obviously, many types of products and an enormous range of applications are encompassed by this paradigm, all driven by human ingenuity, engineering creativity and marketing expertise. To one extent or another, people everywhere are already beginning to enjoy the first wave of benefits of the concept’s reality.
These are the products where this presentation applies (all of them).
A New Interface to Embedded Systems

Gathering data from a running embedded system can be highly difficult. By making this task easier, Micrium’s µC/Probe can substantially enhance your development efforts.
Agenda

- Introduction
- Seeing Inside Embedded Systems
- µC/Probe Features
- µC/Probe from the User’s Perspective
- How µC/Probe Works
- When to Use µC/Probe
- µC/Probe Demo
**Key Takeaways**

By the end of this session you will...

- Be familiar with the capabilities of Micrium’s µC/Probe
- Have a basic understanding of how µC/Probe works
- Be prepared to use µC/Probe in your own projects
Introduction
Micrium at a Glance

- Founded in 1999 by Jean Labrosse, developer of µC/OS, µC/OS-II, and µC/OS-III
- Headquarters in Weston, FL (USA), with a second office in Montréal, QC (Canada)
- Mission is to provide high-quality, well-documented software to the embedded community
  - Micrium is truly "for the way engineers work"
  - With Micrium’s products, embedded software developers have a clear time-to-market advantage

Micrium was started by Jean Labrosse, who was the primary developer of both of the kernels that the company currently offers, µC/OS-II and µC/OS-III. The predecessor of these two kernels, µC/OS, was also written by Jean. Now a familiar face to many in the embedded community, Jean has been developing embedded systems for decades. He has written numerous books and articles describing his work, and he is a regular speaker at many embedded trade shows.

Jean originally wrote µC/OS due to problems that he experienced with commercial kernels. Jean was employed by Dynalco Controls at the time, and his kernel was first intended to be used in his projects for the company. Based on a few articles that Jean wrote, though, his kernel soon found its way into the hands of thousands of developers. By the time Jean introduced the follow-up to µC/OS, µC/OS-II, in 1998, the job of maintaining his software and responding to the phone calls and e-mails of the numerous developers who were using it had become overwhelming. Thus, in 1999, Jean founded Micrium.

Jean’s company has grown substantially since its inception. Currently, Micrium has two separate offices; the company’s headquarters are located in South Florida, while a satellite office is located in Jean’s native Canada, in Montreal.

Micrium is often described as a provider of high-quality embedded software, and quality is what separates the company from other vendors. Micrium’s engineers follow stringent coding standards and write thorough documentation for all of their software. This attention to detail accounts for Micrium’s popularity amongst engineers. Micrium’s software makes their jobs easier and helps them get to market quickly.
Micrium’s Software Components

- µC/OS-II and µC/OS-III
  - Real-Time Kernels
- µC/OS-MMU and µC/OS-MPU
  - Kernel add-ons
- µC/FS
  - File system module
- µC/GUI
  - Graphical software for embedded systems
- µC/TCP-IP
  - Highly dependable TCP/IP stack

Micrium is best known for real-time kernels. Many developers were introduced to Micrium though µC/OS-II, a highly popular real-time kernel that was first released over ten years ago. The follow-up to µC/OS-II, the aptly named µC/OS-III, made its debut in March 2009. This robust kernel offers round-robin scheduling and support for an unlimited number of application tasks. Both µC/OS-II and µC/OS-III are thoroughly described in books from Jean Labrosse.

Developers can opt to supplement µC/OS-II or µC/OS-III with a couple of add-on modules currently available from Micrium. These modules, µC/OS-MPU and µC/OS-MMU, allow the kernels to utilize the memory management hardware that is becoming increasingly common in 32-bit microcontrollers. In the absence of the modules, the kernels simply use a flat memory map. Developers who couple either of the kernels to an add-on module can gain hardware protection from memory corruption.

After the kernels, one of Micrium’s most popular products is µC/FS. This file system module, written entirely in ANSI C, provides support for both the FAT file system and Micrium’s own Embedded File System (EFS). µC/FS drivers are available for practically every popular form of storage device, including NAND Flash, SD Cards, and even USB drives.

Developers seeking to take advantage of a graphical LCD can turn to Micrium’s GUI module, µC/GUI. This highly reliable module provides a multitude of helpful services. Developers can utilize these services on an extensive selection of different LCD controllers, thanks to the large number of µC/GUI drivers available.

µC/TCP-IP is Micrium’s TCP/IP stack. This module was designed entirely from scratch, yet it allows developers to make use of the popular BSD socket interface. A book describing the TCP/IP stack will be released soon.
Micrium’s USB stacks, µC/USB Device and µC/USB Host, afford developers a straightforward means of adding USB capabilities to a product. The stacks are highly efficient and both feature support for a number of popular USB classes, including Mass Storage and Communications Device. The stacks have been run on numerous different USB controllers.

µC/CAN and µC/Modbus round out Micrium’s collection of protocol stacks. Like Micrium’s other stacks, these modules are exceedingly dependable and can easily be ported to new hardware.

Other products available from Micrium include an easy-to-use bootloader (µC/FL), a shell (µC/Shell), a module capable of generating CRCs (µC/CRC), a clock module (µC/Clk), and an interface for character LCDs (µC/LCD).
Seeing Inside Embedded Systems
Developers trying to determine whether or not their code is running as expected often turn to LEDs. These components can be manipulated with relatively little code, and most evaluation boards include at least one or two of them. LEDs, then, are a low-cost method for gaining visual feedback from an embedded system.

Although the feedback provided by LEDs can prove useful to developers, a blinking light hardly constitutes a wealth of information. Using LEDs, developers can see which portions of their application code are being executed, but other, more advanced diagnostics cannot easily be performed. LEDs are ill-suited for displaying the values of variables, for instance.

Another problem with LED use is that this form of obtaining feedback from embedded systems requires developers to instrument their application code. If a developer wants an LED to brighten whenever a particular function is executed, new code must be added to that function. In most cases, this code is pollution; it is not supposed to be present in the completed system. Even when such pollution involves just a few lines of extra code, it can cause problems.
printf() is another means of obtaining feedback from embedded systems. With printf(), developers can display the contents of memory buffers, the values of error codes, the results of analog-to-digital conversions, and other important information. In order for them to do so, however, their software must include drivers for printf() and their development environment must entail some sort of console for viewing printf() output.

Unlike the code associated with LEDs, that associated with printf() cannot be considered trivial. This code includes both drivers and the function itself. In some development environments, the addition of a single printf() call to an application can bring about an increase in code size of as much as 10 kBytes. An application’s RAM footprint can also increase substantially as a result of printf().

Larger memory footprints are not the only side effect of printf() usage; application performance can also be affected. Typically, any drops in performance are only noticeable in debugging, since completed systems generally do not utilize printf(). Even these changes can be harmful, though. They can actually create new bugs or mask existing ones. This phenomenon is sometimes referred to as the Heisenberg Effect.

The Heisenberg Effect aside, unnecessary printf() calls are pollution, and developers often need to generate an excessive amount of this sort of pollution in order for printf() to provide a comprehensive view of their systems. Thus, even on high-performance platforms with abundant memory resources, the use of printf() can be problematic.
Graphical Displays

- Feedback can take the form of words or pictures
- Application code must be instrumented
- Support code is not trivial

Developers dealing with complex systems often find graphical feedback to be more helpful than text. The graphical LCDs present on some hardware platforms are one means of obtaining such feedback.

Although the information provided by an LCD can prove highly beneficial, the use of a display for monitoring an embedded system can cause many of the same problems that are associated with LEDs and printf(). For instance, graphical displays, when used for monitoring, are a source of code pollution. Developers must add code to their application whenever they wish to display new data. Since LCDs typically serve as user interfaces, not diagnostic tools, in completed systems, this extra code must eventually be removed.

Graphical displays also necessitate drivers, and these drivers can be highly complex. Accordingly, displays can be a source of the Heisenberg Effect. Even in systems that will ultimately employ display drivers as part of a user interface, the utilization of these drivers for debugging can introduce unforeseen problems.
Many developers turn to a debugger for feedback from their embedded systems. A variety of information can be gleaned from a typical debugger, including the values of variables. These values are usually listed in what is known as a watch window. Different tools offer slightly different versions of the watch window, but, in most debuggers, it is little more than a table of variables and their values.

A key limitation of watch windows is that they are not continuously refreshed. Whereas a printf() call placed in an infinite loop can provide regularly updated information from a running embedded system, a watch window essentially provides static information. The values shown in a watch window are only updated when code is not running. Thus, watch windows force developers to set breakpoints and halt their systems in order to view the effect of a particular piece of code on a variable.

Frequent stopping and restarting of code is a clear example of the Heisenberg effect. There are many types of problems that simply cannot be addressed with conventional debuggers because these tools change application behavior so drastically. A bug involving the enumeration process in a USB device, for example, would be difficult to diagnose with a conventional debugger. Once a device performing enumeration has been stopped, the USB host to which it is attached is likely to abort the process. In order to efficiently debug USB enumeration and a variety of other operations that are common in embedded devices, developers require a tool that can gather information from running embedded systems.
µC/Probe

- Continuous feedback is provided through a rich graphical interface
- Application code does not need to be instrumented
- No special hardware needed
- Little or no impact on the performance of the target system

µC/Probe, which is based around a Windows application that was developed by Micrium, provides developers with valuable feedback from their embedded systems without causing many of the problems that LEDs, graphical displays, debuggers, and printf() create. With µC/Probe, developers can easily display data on a wide assortment of graphical components. The tool allows these components to be combined into a custom user interface.

No code instrumentation is required in order to take advantage of a custom interface created with µC/Probe. Special debugging hardware is likewise not required. Typically, developers need very little time to set up the tool.

µC/Probe’s hardware and software requirements (or lack thereof) are not the only way that the tool differs from conventional debuggers. Unlike practically every other debugger, µC/Probe can gather data from running systems. Furthermore, it can usually do so without negatively impacting application performance. In other words, the Heisenberg effect need not be a major concern for µC/Probe users.
µC/Probe Features
Reading Variables

- Access to any variable (with the exception of local or automatic variables)
- Variables can easily be selected for monitoring via a list
- Values can be displayed graphically or as text

µC/Probe gives developers quick access to variables. The tool can be used to monitor the value of any global variable, meaning any variable not declared within a function. A developer monitoring an embedded system with µC/Probe can find the names of all of that system’s global variables in a list furnished by the tool. By simply checking boxes located beside the names in this list, the developer can indicate which variables should actually be displayed by µC/Probe. Variables can be displayed on a wide range of different graphical components or can be shown as text.
Recording Data

- Any variable that can be monitored can also be logged
- Data is logged into a simple text file
- Speed at which data is logged depends on µC/Probe’s connection to the target

Oftentimes, developers find it helpful to review all of the values that a variable has assumed over a particular period of time. To make such information easy to obtain, µC/Probe facilitates data logging. The tool’s logging capabilities extend to any variable that can be monitored.

Currently, the logging feature makes use of a simple text file. Developers simply specify the name of this file and µC/Probe fills the file with data. The rate at which µC/Probe actually gathers data from an embedded system is determined by the tool’s connection to that system. As subsequent slides will reveal, µC/Probe supports multiple communication protocols, and each of these protocols offers a different maximum throughput.
Writing Variables

- Variables can be written as the target runs
- Text-based and graphical means of manipulating variables are available

Any variable that developers are able to read using uC/Probe can also be written by the tool. Like reading, writing can be accomplished by a variety of graphical components. Developers can also write variables by simply typing a value in a field. No matter which method is used, uC/Probe always writes variables while code is actually running.
Developers using uC/Probe can read and write variables through a plethora of different graphical components. Amongst the available components are gauges, graphs, numerical indicators, and dials. uC/Probe allows such components to easily be combined into a complete user interface.

Normally, developing a user interface for an embedded system requires a substantial amount of programming. In uC/Probe, however, a user interface can be created by simply dragging and dropping components. Developers can use the tool to create professional-quality displays in just minutes.

A significant difference between a uC/Probe-based display and a conventional user interface is that uC/Probe takes advantage of the computing power of PCs. Even highly complex displays created with uC/Probe require few resources from the embedded systems to which they are attached. Using uC/Probe, developers can add a rich, graphical interface to embedded platforms that could not accommodate a conventional user interface.
Non-Intrusive Debugging

- μC/Probe gathers data from running embedded systems
- Effects on the behavior of application code are minimal
- Changes to a user interface created in μC/Probe do not necessitate changes to application code

None of the services provided by uC/Probe require developers to stop the code running on their embedded systems. Furthermore, developers using these services are not forced to make regular changes to this code. Once uC/Probe has been set up for use with a particular target, a graphical interface can be created for that target without any programming.

Although, in most instances, code must be added to an embedded platform to allow it to communicate with uC/Probe, the size of this code is constant and the introduction of the code generally has little effect on performance. In multi-task applications, uC/Probe-related can be placed in a low-priority task so that higher priority tasks can continue to meet their deadlines even as the tool is gathering data. Because uC/Probe’s effects on the behavior of application code are minimal, the tool allows developers to detect problems that more invasive debugging techniques might miss.
Support for Almost Any MCU

- μC/Probe was designed to be a universal tool
- Micrium’s customers have used μC/Probe with a wide variety of hardware platforms
- CPU architecture is not an issue

μC/Probe is a universal tool. The modicum of embedded code that is provided with μC/Probe is highly portable. This code is written entirely in C, so it can be used with essentially any CPU architecture. On most systems, the only hardware-specific code required by μC/Probe is a single device driver. Additional information about device drivers will be provided on the next slide.

Micrium’s customers have used μC/Probe with a multitude of different hardware platforms. Many of these customers obtained all of the code needed to run μC/Probe, including device drivers, from Micrium. Code allowing μC/Probe to run on some of Renesas’s most popular evaluation platforms has been developed by Micrium’s engineers and is available for customer use.
Popular Communication Protocols

- µC/Probe can connect to a target via RS-232, USB, or TCP/IP
- Device drivers are required on the embedded side
- No special debug hardware is needed

uC/Probe supports multiple communication protocols, including RS-232, USB, and TCP/IP. The tool can be used with any embedded system that is capable of communicating via one of these protocols; special debug hardware is not necessary.

Generic communication code is provided with uC/Probe. Generally, this code must be supplemented with device drivers, which are, in some cases, available from Micrium. For USB and TCP/IP communication, a protocol stack is also necessary.

Of course, systems that incorporate Ethernet or USB hardware generally require protocol stacks regardless of whether the systems’ developers plan to utilize uC/Probe. Since uC/Probe’s generic communication code can be used with practically any protocol stack, the tool is a good fit for such systems. Of course, the tool is also well-suited for any systems that have RS-232 capabilities.
Compatibility with Other Tools

- µC/Probe accepts an executable file as input
- Any compiler that produces executable files of the ELF format can be used with µC/Probe
- In many cases, µC/Probe can be used simultaneously with other tools

µC/Probe is intended to be a supplement for conventional IDEs. The executable files output by an IDE are actually used by µC/Probe as input. These files allow the tool to create lists of variables and to associate addresses with the variables in the lists.

Executable files must be of the ELF format in order for µC/Probe to recognize them. Most popular development environments for embedded systems output ELF files by default, and many of those that do not at least give developers the option of generating this type of file. Thus, IDEs that cannot be used in conjunction with µC/Probe are rare. There are many IDEs that developers can actually run simultaneously with µC/Probe.
Kernel Awareness

- µC/Probe is a perfect match for Micrium’s newest kernel µC/OS-III
- µC/OS-III data screens are provided with µC/Probe
- Since it is a universal tool, µC/Probe is not dependent on any kernel

Since µC/Probe accepts ELF files as input, the tool can be used in conjunction with practically any embedded software written in C. Although Micrium is known as a provider of real-time kernels, µC/Probe does not require developers to build their applications around a kernel. Simple foreground/background applications that contain the requisite communication code can be used with the tool.

Many of Micrium’s customers, of course, use µC/Probe with Micrium’s newest real-time kernel, µC/OS-III. A user interface for this kernel is actually provided with the tool. In other words, µC/Probe offers kernel awareness for µC/OS-III. Developers can use the kernel awareness capabilities to easily monitor the status of tasks in their µC/OS-III-based applications.
Micrium offers two different uC/Probe evaluation packages. One of the packages allows developers to test drive the software for 30 days. Developers with this package can make use of all the same features that are available to uC/Probe licensees. The other evaluation package does not impose a time limit on users but allows no more than 5 different variables to be read or written by any uC/Probe workspace.

One particular workspace is actually exempt from the 5-symbol limit: the uC/OS-III kernel-awareness workspace. This workspace reads a large number of variables, but developers can still open and run it with the 5-symbol version of uC/Probe. Essentially, then, the evaluation software affords developers an entirely free uC/OS-III kernel-awareness tool.
μC/Probe from the User’s Perspective
There are two basic tasks that developers can carry out using uC/Probe: creation of user interfaces and interaction with embedded systems through those interfaces. uC/Probe presents developers with two different views that correspond to these tasks: Design View and Run-Time View.

Design View is what developers see as they put together user interfaces. In this view, graphical components can be dragged and dropped onto data screens and variables can be associated with these components. Some components allow variables to be read, while others facilitate writing.

The components placed on a data screen become active when uC/Probe’s Start button is clicked. The Start button causes the tool to enter Run-Time View. A return to Design View can be initiated via the Stop button.
The canvas to which uC/Probe users add graphical components in order to create a user interface is called a data screen. Multiple data screens can be saved as part of a project, and multiple projects can, in turn, be combined within a workspace. Developers can navigate through a workspace’s data screens and projects via the Workspace Explorer that appears on the left-hand side of the main program window.

Below the Workspace Explorer, in the bottom left-hand corner of the main program window is uC/Probe’s Symbol Browser. Via the Symbol Browser, an ELF file can be associated with a uC/Probe workspace. Once this action has been taken, the Symbol Browser displays a list of all the variables contained in the program that was used to generate the ELF file. Developers can associate a variable with a component by simply checking a box in the list.

The numerous components available to uC/Probe users can be accessed via the toolbars that grace the top of the main program window. Each type of component has its own toolbar. Adding a component to a workspace is as easy as dragging that component from the appropriate toolbar.

Developers can begin to actually use a completed data screen by clicking the Start button. This button is located in the upper left-hand corner of the main program window. Beside it is the Program Options circle, which allows developers to make changes to the uC/Probe environment and to configure the tool for a particular communication protocol. Additional information on configuring uC/Probe will be provided on subsequent slides.
A uC/Probe data screen can take on countless different appearances, thanks to the variety of different graphical components available to the tool’s users. The number of components offered by uC/Probe has grown substantially since the tool was first introduced, and new components will undoubtedly be added in the future.

Although the tool’s graphical components probably attract the most attention, uC/Probe also provides developers with text-based mechanisms for reading and writing variables. Spreadsheets, offering many of the same features as those in Microsoft Excel, can be added to any uC/Probe workspace. Like a graphical component, any field in a spreadsheet can be associated with a variable.
The large number of components available in uC/Probe is only one of the reasons that the tool’s data screens can take on such a wide range of appearances; another important reason is the configurability of the components. Developers can configure any uC/Probe components, including the spreadsheet component, to meet the requirements of a particular project.

A component’s configurable parameters can be accessed by simply clicking the component and then selecting the Properties tab in the Workspace Explorer. The contents of the Properties tab varies from component to component. Typical configurable parameters, though, include color, maximum and minimum value, and resolution.
The uC/Probe environment, like the components available through that environment, can be configured to better serve developers’ needs. The Options dialog box, which can be reached via the Program Options circle, allows developers to adjust a number of different parameters. The uC/Probe environment can be set up through this dialog, as can the communication link connecting the tool to an embedded system.

Communication parameters must be established before uC/Probe’s Run-Time view can be used. Via the Options dialog, developers can select from any of the different protocols that the tool supports and can then adjust the parameters of the chosen protocol. A developer seeking to use RS-232, for example, can select that protocol on the dialog’s Communication page and can then use the RS-232 page to select a COM port and baud rate. Similar procedures can be used to establish TCP or USB communication. An indication of which protocol the tool has been set up to use is provided in the status bar that appears at the bottom of the main program window in Design View.
How µC/Probe Works
Implementation Overview

- μC/Probe is made up of two components
  - A Windows program that serves as a customizable interface to running embedded systems
  - Embedded code that responds to requests from the Windows program

The most visible part of uC/Probe is the Windows application through which developers create user interfaces to interact with embedded targets. A portion of uC/Probe, though, actually resides on those targets. The embedded portion of uC/Probe is responsible for responding to requests sent from the Windows application. The Windows application can either request that a particular memory location be written or that the value currently stored at a particular location be returned.
Before uC/Probe can begin communicating with an embedded system, some sort of user interface for that system must be fashioned in the tool’s Design View. During the creation of a user interface, graphical components are dragged onto data screens and then, via the Symbol Browser, are associated with variables. Only components that have been associated with variables can be updated by uC/Probe.
Communicating with the Target (Cont.)

- The ELF file supplied to µC/Probe as input lists the address of each variable

Within µC/Probe, components are associated not only with variables but with addresses. µC/Probe is able to determine the address of a given program’s variables by parsing that program’s ELF file.
Communicating with the Target (Cont.)

- μC/Probe uses these addresses to formulate read and write requests

uC/Probe uses addresses gathered from the ELF file to formulate requests for the embedded portion of the tool.
Each request created by the Windows application is sent to the attached embedded system. Which protocol is used to send the requests depends on the settings made in the Options dialog.
Communicating with the Target (Cont.)

- µC/Probe updates graphical components based on the target's responses to the requests

On the embedded side, the reception of a packet always results in a memory location being either read or written. In the case of a read, the contents of the memory location are returned to the Windows application in a separate packet. Using these responses, the Windows application updates graphical components so that they indicate the current values of the variables that they represent.
The code that must be present in an embedded system in order for uC/Probe to be used with that system can, like the tool itself, be divided into two parts. One of these parts comprises the generic, hardware-independent communication code that is provided with uC/Probe, and the other part includes device drivers and protocol stacks. Since TCP/IP and USB stacks are far from trivial pieces of software, the latter part can have substantial memory requirements. In most systems, though, this code is not used by uC/Probe alone. The code that is specific to uC/Probe – the generic communication code – is compact, usually requiring only a few kBytes of memory space.
Update Rates

- The speed at which requests are sent from the PC to the target varies according to communication protocol.
- Via µC/Probe’s Options dialog, the delay between requests can be adjusted.

<table>
<thead>
<tr>
<th>Communication Method</th>
<th>Transmission Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS-232</td>
<td>1,000 symbols/second</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>11,000 symbols/second</td>
</tr>
<tr>
<td>USB</td>
<td>4,000 symbols/second</td>
</tr>
</tbody>
</table>

The communication protocols supported by uC/Probe differ substantially in terms of maximum data rate. TCP/IP, for example, can support much higher rates than RS-232. The differences in the protocols’ capabilities means that not all uC/Probe workspaces are updated at the same rate; update rates vary with communication protocol.

Update rates in uC/Probe are also affected by user-supplied options. Via the options dialog, time delays can be inserted between requests. Such delays can help developers to limit the CPU usage of the embedded portion of uC/Probe.
When to Use µC/Probe
Debugging

- μC/Probe can be used to supplement a conventional debugger
  - Especially helpful for detecting stack overflows

- Kernel awareness aids in debugging μC/OS-III-based applications

<table>
<thead>
<tr>
<th>Task Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Used</td>
</tr>
<tr>
<td>(Entries)</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>28</td>
</tr>
<tr>
<td>36</td>
</tr>
<tr>
<td>36</td>
</tr>
<tr>
<td>34</td>
</tr>
</tbody>
</table>

μC/Probe is typically seen as a debugging tool, and it is indeed well-suited for helping developers to identify problems in their code. Developers can use μC/Probe alongside a conventional debugger to significantly increase their chances of catching a variety of bugs. Some traditional sources of problems in embedded systems, such as stack overflows, are easier to detect with μC/Probe than with conventional tools.

Although μC/Probe can be used to detect bugs in practically any code written in C, the tool is particularly well-suited for debugging μC/OS-III-based applications. μC/Probe’s kernel-awareness module allows developers of such applications to sight deadlocks, task starvation, and other problems that are specific to multi-task code. Since the kernel-awareness module is actually just a collection of data screens capable of displaying μC/OS-III variables, developers of kernel-based applications can debug their code using the same procedures that are followed by developers whose code does not make use of μC/OS-III.
In addition to offering unique debugging capabilities, uC/Probe affords developers with an ideal platform for creating demos and presentations. With uC/Probe, visually appealing demos can be crafted for embedded systems having no display hardware whatsoever. By simply dragging graphical components onto a uC/Probe data screen, a developer can put together a professional-quality demo in a matter of minutes.

Unlike traditional demos, those created with uC/Probe are highly portable. A single uC/Probe demo can actually be used with a wide selection of different embedded systems, so long as the variables associated with the demo’s graphical components exist in each system. Accordingly, the tool is ideal for developers who prepare demos and presentations on a regular basis.
uC/Probe’s wide selection of highly configurable graphical components allows developers to easily give their demos a custom look. Customization of demos is also facilitated by uC/Probe’s support for user-supplied text and images. Like the tool’s graphical components, custom text and images can simply be dragged onto any data screen.
In the Field

- μC/Probe is not solely a development tool
- Technicians can use μC/Probe to gather status information from a product
- Product performance is usually not affected by target-resident code

μC/Probe’s benefits actually extend beyond the development process. Since the tool supports communication protocols that are often used by finished products, it can serve as an aid for technicians responsible for troubleshooting such products. The non-invasive manner in which μC/Probe interacts with embedded systems increases the tool’s usefulness in such post-deployment situations.

In the future, μC/Probe will be given even more features helpful to technicians who use the tool with finished products. The tool’s debugging and demonstration capabilities will also expand. Within Micrium, a team of engineers is dedicated to maintaining and improving μC/Probe, so the embedded community can expect to see new versions of the tool released on a regular basis. μC/Probe has already won multiple awards and helped numerous developers successfully complete challenging projects, but this unique tool’s future looks even brighter than its past.
µC/Probe Demo
Questions?
Innovation
Thank You!