Developing a Debugger for Real-Time Operating System

[1, 2] *A.N. Emelenko <emelenko@ispras.ru>*
[1, 3] *K.A. Mallachiev <mallachiev@ispras.ru>*
[1, 2, 3] *N.V. Pakulin <npak@ispras.ru>*
[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*
[2] *Moscow Institute of Physics and Technology (State University), 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia*
[3] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia.*

Abstract. In this paper, we report on the work in progress on the debugger project for real-time operating system JetOS for civil airborne systems. It is designed to work within Integrated Modular Avionics (IMA) architecture and implements ARINC 653 API specification. This operating system is being developed in the Institute for System Programming of the Russian Academy of Sciences and next step in developing this system is to create a tool to debug user-space applications on it. We also discuss the major requirements to such a debugger and show the difference between it and typical debugger, used by desktop developers. Moreover, we review a number of debuggers for various embedded systems and study their functionality. Finally, we present our solution that works both in emulator QEMU, which we use to emulate environment for our system, and on the target hardware. The presented debugger is based on GDB debugging framework but contains a number of extensions specific for debugging embedded applications. However, the implementation of the debugger is not complete yet and there is a number of features that can improve debugger usability, but it is already more functional than common GDB debugger for QEMU and, in contrast to other systems and their debuggers, where developers can use some functions to debug applications, but not all we need, our debugger meets the majority of our requirements and restrictions.

Keywords: debugger; GDB; real-time OS; remote debugger

DOI: 10.15514/ISPRAS-2016-28(2)-13

For citation: Emelenko A.N., Mallachiev K.A., Pakulin N.V. Developing a Debugger for Real-Time Operating System. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 2, 2016, pp. 193-204. DOI: 10.15514/ISPRAS-2016-28(2)-13

1. Introduction

Application debugger is an indispensable tool in developer's hands. But debugger in a real-time operating system is more than just plain debugger. In this paper we present an on-going project on debugger development for JetOS, a real-time operating system that is being developed in the Institute for System Programming of the Russian Academy of Sciences.

JetOS is a prototype operating system for civil airborne avionics. It is designed to work within Integrated Modular Avionics (IMA) architecture and implements ARINC 653 API specification, the de-facto architecture for applied (functional) software.

The primary objectives of ARINC 653 are deterministic behavior and reliable execution of the functional software. To achieve this ARINC 653 imposes strict requirements on time and space partitioning. For instance, all memory allocations and execution schedules are pre-defined statically.

The unit of partitioning in ARINC 653 is called *partition*. Every partition has its own memory space and is executed in user mode. Partitions consist of one or more *processes*, operating concurrently, that share the same address space. Processes have data and stack areas and they resemble well-known concept of threads.

Embedded applications might be run in two different environments: in an emulator and on the target hardware. In our project we use QEMU system emulator. Although QEMU has its own debugger support, its functionality proved to be insufficient for debugging embedded applications. Therefore, we implemented a debugger not only for the target hardware, but for the emulator as well.

2. Main Targets for Debugger

Debugger for an embedded operating system has a number of specific features compared to typical debugger used by desktop developers.

Firstly, an embedded application runs under constrained conditions, such as limited on-board resources and lack of interactive facilities – no keyboard and screen. This makes it impossible to do debugging on the same device where application runs. Therefore, the debugger for embedded applications has to be remote: the developer interacts with workstation while the application runs on a target hardware.

Secondly, an embedded application typically consists of a number of interacting processes that needs to be debugged simultaneously. This means that the debugger must support dynamic and transparent switching between execution contexts during debugging session.

Thirdly, the debugged should support developers of system software, mostly device drivers and network stack. This requires switching between low-privilege code and highly privileged kernel code in the same debugging.

It is also important to mention that embedded developers widely use emulators in their work process. Typically most of development runs on top of emulators, therefore the debugger must support corresponding emulators as well.

The above mentioned features impose a number of restrictions on the design of the debugger that we considered:

- There are many different applications compiled in OS, which can have overlapping virtual address spaces.
- Typically target hardware board for embedded OS has only one port to communicate with the external world – a single serial port. Since it is used to stream console output of the running applications we need to share it between debugger traffic and applications' output.
- Multifunctional debugger is a complex program. It is very complicated to develop it from scratch, so we decided to base our debugger on an existing one.
- Support debugging both on hardware and with emulator because this support can expand developers' capabilities and improve their efficiency.
- Support capabilities of debugging for kernel and for user mode code, as well as capabilities of multiprocess mode.
- It must excel QEMU debugger, which we use to emulate environment for our system.
- Since the OS in question is real-time, it is important to minimize debugger's impact on system during debugging.

In order to meet these restrictions we selected the architecture of remote debugger with server and client parts, that communicate over a serial port using multiplexer.

We have chosen GDB (GNU debugger) for the client part of our debugger.

3. Related Works

We are not the first to consider the problem of remote debugging. For example, Pistachio microkernel uses kdebug for debugging [4]; besides, there is Fiasco debugger [1] and many different debuggers for VxWorks, for example, RTOS debugger [2].

Here we briefly consider some debuggers for embedded OSes and their primary features.

3.1 Fiasco OS

Fiasco OS is a 3rd-generation microkernel, based on L4 microkernel [1]. The kernel is simplistic, it misses most of the features available in “big” operating systems like Linux or Windows: program loading, device drivers and file system. All these features must be implemented in user-level programs on top of it (L4 Runtime Environment provides a basic set such functions).

Fiasco OS has built-in support for debugger that:

- supports threads;
- provides stack backtrace

- sets breakpoints;
- does single step;
- provides reading/writing in memory;
- provides reading hardware registers;
- support interprocess communication (IPC) monitoring.

The Fiasco Kernel Debugger (JDB) is a debugger for Fiasco. It has the following special functionality:

- It always freezes the system when it is working. It means that JDB disables all interrupts and halts clock. All processes and kernel don't work when JDB is invoked.
- JDB doesn't use any part of Fiasco kernel, because it is a stand-alone debugger with drivers for keyboard, display, etc.

In general, JDB is not a part of Fiasco µ-kernel, and Fiasco µ-kernel can run without connection with JDB or another debugger.

The debugger operates remotely over the serial line.

3.2 VxWorks

VxWorks [5] is a real-time operating system (RTOS) developed as proprietary software by Wind River of Alameda, California, US. It supports Intel (x86, including the new Intel Quark SoC and x86-64), MIPS, PowerPC, SH-4, and ARM architectures.

RTOS debugger for VxWorks implements the following set of features:

- Task Stack Coverage
- Task Related Breakpoints
- Task Context Display
- Debugging Modules (for example, Kernel module)
- Debugging Real-Time Processes
- Debugging Protection Domains
- Collecting statistics for function and tasks

RTOS debugger displays all system states, tasks, message queues, memory partitioning, modules and etc.

The key feature of the RTOS debugger is that is based on Lauterbach's TRACE32 debugger [3] that utilizes hardware interfaces like JTAG. It does not use serial port for communication with the target hardware but rather requires specific debug module.

3.3 L4Ka::Pistachio

L4Ka::Pistachio [4] is the latest L4 microkernel developed by the System Architecture Group at the University of Karlsruhe. It is the first available kernel

implementation of the L4 Version 4 kernel API, which provides support of both 32-bit and 64-bit architectures, multiprocessor and super-fast local IPC. The current release supports x86-x64 (AMD64/ EM64T, K9 / P4 and higher), x86-x32 (IA32, Pentium and higher), PowerPC 32bit (IBM 440, AMCC Ebony / Blue Gene P).

The debugger for Pistachio kernel can direct its I/O via the serial line or the keyboard/screen. It is a local debugger and does not support remote debugging mode.

This debugger is also a low-level device with very limited amount of functions.

Debugger for Pistachio can:

- Set breakpoints
- Single step
- Dump memory
- Read registers

When the processor meets special instruction (for example, *int3* instruction), it passes control to interrupt handler, which is the part of Pistachio kernel. In turn, interrupt handler checks instructions, which come next, and if they correspond to the special layout, it prints special message before passing control to interrupt handler. This feature is a simplistic implementation of a facility to trace execution.

4. Technical Description:

The primary goal of the debugger is Power PC platform, based on e500mc CPU core. The debugger is based on GDB, it uses the GDB architecture to establish link to the remote target.

The architecture includes three major components: front end, local client and remote server. The front end provides user interface, it runs on the same workstation as the client part. The latter translates the commands from the front end into GDB protocol and communicates with the remote server. The server implements the actual command embedded into protocol messages such as reading memory regions, setting breakpoints, processing debug interrupts, etc. Remote server is sometimes called “stub”.

Gdb-stub for i386 was taken as a basis for our debugger. This stub was totally redesigned for e500mc processor, which belongs to PowerPC architecture family. We left only the packet exchange and some of the packet processing mechanisms.

We use common gdb client, which was built for PowerPC with somewhat extended functional, to connect to our stub. This functional was developed using special user defines commands, so developers don't need to use special version of GDB. Instead, they can use any version, but it needs to use gdb commands file by utilizing special “source” command in GDB.

Accordingly, messaging mechanism between client and server doesn't change – the client sends a special-type packet to the server and waits for the server's answer. The server receives this message, checks control sum, which was sent in this packet, and if it matches the message contents, informs the client that the message was accepted

for processing. Then the server performs the action described in the packet and sends its own packet to the client.

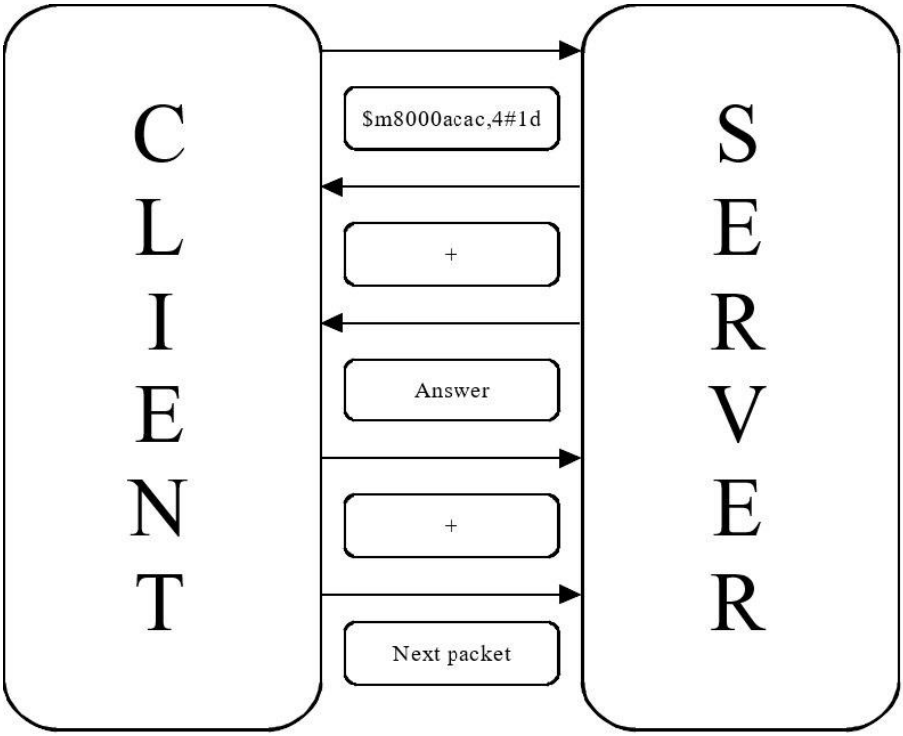

Fig. 1. GDB messaging mechanism

Let us consider an example on Fig. 1. Here client sends to server packet "$m8000acac,4#1d". This means that client wants to read 4 bytes of memory from virtual address 0x8000acac. In this packet "1d" is the control sum, that is, the sum of all bytes in message modulo 256. If the server fully receives this message, it sends "+", and the client knows that the message was accepted. After that, server sends 4 bytes of memory from that address to the client in the same way, and message exchange continues. All these types of packets are described in GDB manual.

4.1 Implementation of the server side

In general, debugger's work consists of packet exchange between client and server. Client sends certain types of packets to perform the action, which the user needs. Our goal is to develop server part because we use client part from common GDB.

During the connection between server part of the debugger with client part our system stands in frozen state where no interrupts are available and the clock is halted. This opportunity allows us to work with partitions and debugger as if there is no debugger in the system.

We implemented functions in our debugger in the following way:

Breakpoints setting was implemented using special PowerPC instruction 'trap'. When the trap instruction occurs, server code in interrupt handler is called.

For Single step operation, we can use two different methods. The first one is when the system stops on the next instruction of the current partition. The second one is to stop the system stops on the next instruction wherever it is. The difference is how system calls are handled; the first method skis all kernel code and traverses application only. The second method allows entering kernel and stepping through system call implementation. Furthermore, it is sensitive to interrupts: if an interrupt occurs during the step, the debugger switched to the interrupt handler.

However, GDB structure requires interrupts to be disabled during single step. This requirement imposes restrictions on partition's work, so we gave up the second method. Because of the lack of debug registers in QEMU we need to disable interrupts and set trap instruction on the next instruction.

Watchpoints were implemented using special capabilities of hardware, such as Debug registers. Unfortunately, QEMU doesn't have such registers, so we need to use another way to set watchpoints in emulator. This method isn't implemented yet, but we are working in this direction.We also developed multiplexer to use one serial port for both GDB and another application. Multiplexer allows message exchange for debugger and for internal system service. The transformation of one serial port into two serial ports with the help of our multiplexer is not so difficult.

There are two parts of multiplexer, local and remote. Local part is a superstructure responsible for information input/output in the system. During the output it puts a special symbol before every printable symbol, determining to which of the two virtual serial ports the next symbol should be sent. Working with input symbols is very similar: two symbols are read, with the first of them specifying the application to which we want to send the second symbol. Remote part of multiplexer looks the same. This solution is not the fastest, but it provides smooth debugger's work via one serial port together with other applications. This connection between remote and local parts of multiplexer is shown on Fig. 2.

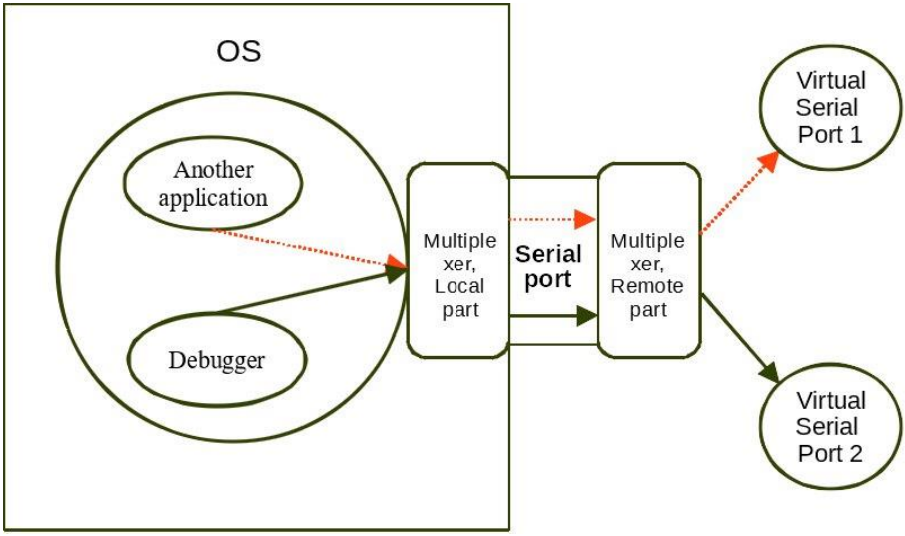

Fig. 2. Multiplexer work

5. Debugger's Capabilities

Our debugger supports all standard debugging features. Among them are:

5.1 Setting Breakpoints on Kernel and Partitions.

Setting breakpoints is the key feature of any debugger. Considering that client knows only virtual addresses, the server part of the debugger must correctly translate this address into physical address. Our debugger can do this, that's why users can debug partitions with overlapping virtual address spaces and debugger stops only on the partition that the user wants.

5.2 Single Step.

Stepping through code step by step is a convenient way of finding bugs. However, there can be a situation in real-time OS, when the next instruction in code is not the next executable instruction, for example, because of timer interrupt. That's why we disable interrupts during the single step.

5.3 Showing Information about Processes and Threads, Inspecting Memory, Instructions and Registers. Memory Reading and Writing.

Memory view must correctly translate virtual addresses into physical as with breakpoints. The capability to find out all information about threads in OS, their states, registers and memory is very important too.

Support of memory writes allows changing process state as user's discretion.

5.4 Setting Watchpoints.

Watchpoints are one of the most comfortable ways to control user's partition. They give the opportunity to follow changes in memory sectors and stop\pause while trying to read or record memory. This opportunity increases the number of ways to control partitions' states.

5.5 Stack Inspection.

Stack inspection makes tracing possible: for example, tracing the queue of called functions, which can help user to understand exactly what has happened in the system.

6. Future Work

Implementation of the debugger is not complete yet. There is a number of features that can improve debugger usability:

- Enhance debugging capabilities to the level of standard GDB functionality.
- Accelerate debugger interaction time with the system through multiplexer.
- Improve hardware support on bare metal.
- Increase user convenience in multiplexer. Enhance its functionality for working with more devices (now multiplexer supports only two devices). This solution allows us to work on bare metal with as many ports as we need, regardless of the actual amount of ports.
- Add watchpoints implementation to QEMU, which doesn't support debug registers. This is the reason why we can't use debug registers for setting watchpoints like we do on bare metal. In that case, we need to change code handling in QEMU to develop instruction for watchpoints creation.

7. Conclusion

In this paper, we have presented our project on implementation of the debugger for real-time operating system JetOS. In contrast to other systems and their debuggers, where developers can use some functions to debug applications, but not all we need, our debugger meets the majority of our requirements and restrictions. However, we will able to update our debugger in near future and increase its functionality, but it is already more functional than common GDB debugger for QEMU.

References

[1]. F. Mehnert, J. Glauber and J. Liedtke, “Fiasco Kernel Debugger Manual” Dresden University of Technology, Department of Computer Science, November 2008 (https://os.inf.tu-dresden.de/fiasco/doc/jdb.pdf)

[2]. Lauterbach GmbH, “RTOS debugger for VxWorks”, November 2015

(http://www2.lauterbach.com/pdf/rtos_vxworks.pdf)
[3]. Lauterbach GmbH, "RTOS-VxWorks", 18 August 2014 (http://www2.lauterbach.com/doc/rtosvxworks.pdf)
[4]. System Architecture Group University of Karlsruhe. "The L4Ka:: Pistachio Microkernel". May 1, 2003 (http://www.l4ka.org/l4ka/pistachio-whitepaper.pdf)
[5]. Wind River Systems, Inc "VxWorks Product Overview", March 2016 (http://windriver.com/products/product-overviews/2691-VxWorks-Product-Overview.pdf)
[6]. Free Software Foundation, Inc. "Debugging with gdb: the GNU Source-Level Debugger", The Tenth Edition (https://software.intel.com/sites/default/files/article/365160/gdb.pdf)

Разработка отладчика для операционной системы реального времени

[1, 2] *А.Н. Емеленко <emelenko@ispras.ru>*
[1, 3] *К.А. Маллачиев <mallachiev@ispras.ru>*
[1, 2, 3] *Н.В. Пакулин <npak@ispras.ru>*
[1] *Институт системного программирования РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский физико-технический институт (государственный университет), 141701, Московская область, г. Долгопрудный, Институтский переулок, д.9.*
[3] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1.*

Аннотация. В этой статье мы расскажем о проекте по разработке отладчика для операционной системы реального времени JetOS, созданной для гражданских авиационных систем. Она предназначена для работы в рамках архитектуры Интегрированной Модульной Авионики (ИМА) и реализует ARINC 653 спецификацию API. Эта операционная система разрабатывается в институте системного программирования РАН, и следующим шагом в ее разработке стало создание инструмента для отладки пользовательских приложений. Также в этой статье будут рассмотрены основные требования к такому отладчику и показана разница между ним и обычным отладчиком, используемым разработчиками настольных приложений. Более того, были рассмотрены другие встраиваемые операционные системы, такие как WxWorks, Fiasco OS, L4Ka::Pistachio и отладчики для них, а также был изучен их функционал. В заключение, мы представим наш отладчик, который может работать как в эмуляторе QEMU, используемом для эмуляции окружения для JetOS, так и на целевой машине. Представленный отладчик является удаленным и построен с использованием структуры GDB, но содержит ряд расширений, специфичных для отладки встроенных приложений. Однако реализация отладчика пока не завершена и существует целый ряд задач по улучшению удобства и возможностей отладчика, но на текущий момент он является уже более функциональным, чем обычный отладчик GDB для QEMU и, в

отличие от других рассмотренных систем и их отладчиков, где разработчики могут использовать некоторые функции для отладки приложений, но не все, что нам нужны, наш отладчик удовлетворяет большинству поставленных требований и ограничений, а также уже используется разработчиками приложений для JetOS.

Ключевые слова: отладчик; GDB; ОСРВ; удаленный отладчик; операционная система реального времени

DOI: 10.15514/ISPRAS-2016-28(2)-13

Для цитирования: Емеленко А.Н., Маллачиев К.А., Пакулин Н.В. Разработка отладчика для операционной системы реального времени. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 193-204 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-13

Список литературы

[7]. F. Mehnert, J. Glauber and J. Liedtke, “Fiasco Kernel Debugger Manual” Dresden University of Technology, Department of Computer Science, November 2008 (https://os.inf.tu-dresden.de/fiasco/doc/jdb.pdf)

[8]. Lauterbach GmbH, “RTOS debugger for VxWorks”, November 2015 (http://www2.lauterbach.com/pdf/rtos_vxworks.pdf)

[9]. Lauterbach GmbH, “RTOS-VxWorks”, 18 August 2014 (http://www2.lauterbach.com/doc/rtosvxworks.pdf)

[10]. System Architecture Group University of Karlsruhe. “The L4Ka:: Pistachio Microkernel”. May 1, 2003 (http://www.l4ka.org/l4ka/pistachio-whitepaper.pdf)

[11]. Wind River Systems, Inc “VxWorks Product Overview”, March 2016 (http://windriver.com/products/product-overviews/2691-VxWorks-Product-Overview.pdf)

[12]. Free Software Foundation, Inc. “Debugging with gdb: the GNU Source-Level Debugger”, The Tenth Edition (https://software.intel.com/sites/default/files/article/365160/gdb.pdf)