Using modularization in embedded OS

[1,2] K.A. Mallachiev <mallachiev@ispras.ru>
[1,2,3] N.V. Pakulin <npak@ispras.ru>
[1,2,3,4] A.V. Khoroshilov <khoroshilov@ispras.ru>
[1] D.V. Buzdalov <buzdalov@ispras.ru>
[1] Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[2] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] Moscow Institute of Physics and Technology (State University)
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4] National Research University Higher School of Economics (HSE)
11 Myasnitskaya Ulitsa, Moscow, 101000, Russia

Abstract. Modern embedded OS are designed to be used in control solutions in various hardware contexts. Control computers may differ in the architecture of the CPU, the structure of communication channels, supported communication protocols, etc. Embedded OS are often statically configured to create an OS image, which intended to be executed on some specific control computer. System integrator usually performs this configuration. Embedded OS are often developed by many companies. Joint development and integration is very complex if OS doesn't support modularity. Support of modularity and component assembly reduces the need of communication among companies during development and integration. This allows customers to create minimal solutions that are optimally adapted to the particular task and hardware platform. Furthermore, customers may be interested in adding their own low level components without OS modification. In this article, we present an approach to building modular embedded solutions from heterogeneous components based on the RTOS JetOS. The mechanism of components binding developed by us allows uniting heterogeneous components from different manufacturers within the same section of the address space. This mechanism allows component developer to independently develop their components. And system integrator can independently from developers configure what component he likes to see in OS image and how components should interact.

Keywords: embedded systems, components, RTOS.

DOI: 10.15514/ISPRAS-2017-29(4)-19

For citation: Mallachiev K.A., Pakulin N.V., Khoroshilov A.V., Buzdalov D.V. Using modularization in embedded OS. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 4, 2017, pp. 283-294. DOI: 10.15514/ISPRAS-2017-29(4)-19

1. Introduction

Embedded operating systems are built to provide specific functionality on specific hardware. Development of a new OS from scratch for every task and hardware is unwise and operating systems are designed to support several CPU architectures and a lot of peripheral devices in a single distribution. Therefore, OS distribution contains many drivers to support a large number of different hardware. Most of the drivers are not needed for correct OS execution on a specific board. Moreover, many embedded systems are aimed to run in restricted environment, for example with limited memory.

Static OS configuration is used in cases when it is known in advance, on which hardware the OS image is going to be executed. OS configuration is commonly performed by the system integrator. They choose OS features suitable for OS task and drivers for hardware. Only chosen parts will get into final OS image. System integrator doesn't change OS source code. Static configuration allows keeping final image small.

Safety-critical systems must be certified. For airborne systems there is a standard for certification called DO-178C [1], where OS kernel must be certified by highest level of reliability. Certification is complex and lengthy process. Small change in one part of system leads to recertification of the whole system.

We develop an open-source real-time operating system for civil aircraft airborne computers called JetOS. JetOS is ARINC-653 [2] compliant, supports static configuration and aimed to DO-178 certification.

ARINC-653 specifies interfaces that RTOS (real-time operating system) should provide to avionics software, also the standard specifies some design constrains to the OS. The most pertinent constraint is that application code is executed inside partitions that are isolated from each other by resources and in time.

To simplify and minimize OS kernel and therefore to simplify OS certification process we moved drivers and some services from kernel to special ARINC-653 partitions, called system partitions [3]. Besides drivers system partition contains services such as network stacks, file systems, logging, etc.

System partitions should be certified as well as the kernel. Certification for highly-critical software requires absence of unreachable code. Usage of static configuration of the system partition allows to static selection of required drivers and services, and therefore getting rid of unused code.

It is common that there are many vendors involved in building a specific embedded solution: OS vendor, BSP vendor, device driver developers, system integrator, etc. When services or drivers they are developing are strongly coupled, developers have to interact a lot.

Therefore splitting system partition to independent isolated components seems to be suitable solution. Each driver and service will be in dedicated component. Each component would have a single developer.

Component should interact with each other. Appearance of fixed interface between components would make component development easer. Moreover fixed interface

can make system flexible. Statically configured component-based system (in our case system partition) can be flexible in several aspects:

- When there are several components implementing the same interface (e.g. several file systems) and system integrator can choose which component will get into final image.
- When there are several components implementing the same interface, and they all can get into final image. System integrator configure on static, which components interact. For example, if there are two file systems, some component would work with one file system and others with the second one.
- When system integrator can add new component between two interacting, if the new component has a suitable interface. This is useful and can be used, for example, to insert traffic analyzer between protocol stack and network card driver.

Another use-case is to reuse a device driver in an applications stack, such as network card driver in the network stack. Isolated into component the same driver code might serve multiple device instances due to different sets of internal states and configuration parameters. All copies of the component share same driver code, so that each component copy would work with assigned device, would make system scalable and flexible.

Certification of system includes, among others, unit and integration tests. Splitting system partition to components makes certification easier. Component-level tests can be run by component developer. And system integrator doesn't need to rerun unit tests, he only needs to run integration tests.e.

2. Related Works

Classical distributed components models like Enterprise JavaBeans, CORBA, Corba Component Model and DCOM [4,5,6] define components and interfaces between them. Models allow substituting one component with the other one with the same interfaces. Components configuration dynamically configured by brokers. This approach is not suitable for embedded systems with static configuration.

Ideas to separate OS appeared long ago in microkernels. Microkernel architecture's [7,8,9] primary goal is to separates OS into independent servers that could be isolated from each other. Servers interact through inter-process communication (IPC). IPC calls are typed and servers with the same interface can substitute one another. But there cannot be two servers with the same interface; therefore this model is not suitable for our tasks too.

VxWorks is a popular embedded operating system. VxWorks board support package (BSP) is divided into components. Components interface is declared in component description language (CDL). BSP developer can construct BSP from existing component and can add their own components. But this system is not flexible; for example, each component has hardcoded in it a list of names of components it interact

with, therefore one component cannot be easily substituted in a configuration with another one with the same interfaces.

We are not aware of any component based model with the following set of features:

- Static configuration,
- Low overhead,
- Flexible configuration (in all aspects from introduction),
- Low mishit probability, when component interact with component it not designed to (runtime addressing checks)

3. Basic Capabilities of Component-Based Model

Our model aimed to have small overhead, so it can be suitable for RTOS. In its raw form, our model assumes that there is a lot of similar code written by component developers in C language. To reduce the amount of hand work we generate helper code, based on configuration files. Language, which is used to write configuration files, can be any declarative language; we use YAML for these purposes.

3.1 Component developer view

Model defines component types and component instances. Each component has a unique component type and assigned implementation and any number of instances. Component type is similar to term "class" from object oriented languages and component instance is similar to "class objects". Component instances share code, but sharing does not apply to some private data, called instance state.

Components interact. The ability of one component to use services of the others is achieved through typed ports. There are two kinds of component ports:

- Input ports, which show that the component provides some functionality. Input ports have assigned handlers implemented by the component, which will be called when some other component calls the interface of the component.
- Output ports, which are used by a component when invokes behavior of another component. The component calls others indirectly, through output ports.

Ports are typed, input port of one component and output of the other one can be connected only if they have the same port type. Port type is called interface. Interface is the set of functions, which input port provides or output port require. Since interface can have several functions, then output port that implements this interface has several assigned handlers, one for each function in interface.

Interface declares as the set of triple of function names, signature, and return types. Example of simple interface declaration can be seen at fig.1.

```
- name: data_sender
  functions:
    - name: send
      return_type: ret_t
      args_type: [int]
```

Fig 1. Data_sender interface with one function ret_t send(int)

Component type declaration contains component name, component instance state structure, and component ports. Output ports are declared as pair of port name and port interface. Input ports are declared as triple (n, I, m): port name n, port interface I, and m a list of pairs of interface function and assigned implementation specified by components function name.

You can see example of component type configuration at fig. 2.

```
name: Filter
state_struct:
  edge: int

input_ports:
  - name: in
    type: data_sender
    implementation:
     send: filter_send

output_ports:
  - name: out
    type: data_sender
```

Fig 2. Component type Filter. Component state contains one field edge. Componet type has single input port called in, port interface is data_sender, fucntion send of data_sender interface is implemented by filter_send function.

During system build configuration files are parsed and corresponding C code is generated:

- C-structure describing component, with name identical to component name. (e.g. structure `Filter` for component Filter)
- Declaration of functions specified in input ports (e.g. declaration of function `filter_send` for component Filter). This declaration enforces naming convention.
- Special function for calling output ports.

Component developers should use only ports to communicate with other components. Direct call of another component might work but is not guaranteed. The component developer is guaranteed only the interfaces. The developer chooses names for ports. Input ports are an entry point to component. Component developer does not use names on input ports. Output ports are used when component should use service of

another component. To call the output port a developer should specify output port name, output port function name, and function arguments. Developer should not assume what real function of which component will be called. You can see an example of calling function from output port at fig. 3

```
ret_t filter_send(Filter *self, int data)
{
  ...
  res = Filter_call_out_send(self, data);
  ...
}
```

Fig 3. Call of function send of port out.

3.2 System integrator view

System integrator decides how many instances of each component should be created, and how they are connected. For each component, they choose unique name, and how to initialize its state. System integrator uses instance names and names of their ports to link ports of different instances. All of this information integrator specifies in configuration file. Graphical view of example configuration use can see at fig. 4.

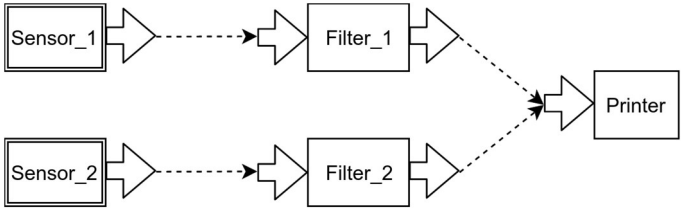

Fig 4. Example linkage configuration. Sensor_1 and Sensor_2 are instances of Sensor component type. Filter_1 and Filter_2 are instance of Filter component type. Sensor_1 ouput port connect to Filter_1 input port. Filter_1 input port connected to Printer. Same for Sensor_2 and Filter_2

4. Advanced capabilities of component based model

4.1 Init function

Instances can have init function: component developers should declare init function name in configuration. At system partition start all init functions of all instances are called sequentially. There is no way to specify dependencies on init (i.e. init of open component should be called before init of the other one) because we assume that components are independent and should not have any dependency.

4.2 Active and reactive components

All components with input ports are *reactive*, i.e. get control by call from other component. Some components are *active*, i.e. the component gets control from OS by some regularities (periodically or by event). Component can be active and reactive at the same time.

There are two types of active components in our model:

- Components which have a special entry point – activity. This type of active components is useful when component instances should do some simple work from time to time (for example, checking whether there are any new networks packets). Component developer declares activity name in configuration. All activities are called sequentially. This type of active components has a big disadvantage: if some instance will freeze in its activity then all instances of this type in the system are going to freeze, so component developer should not use any wait objects in activity.
- Components, which instances create their own threads inside init function. In this case freezing of the instance, which is running in the dedicated thread, will not cause freezing other instances.

4.3 Array of ports

Sometimes component developers need to create configurable number of ports of the same type. We support array of ports, but only for output ports. For calling function of output port array developers should specify index in the array besides port name, function name and function arguments.

Arrays of ports are useful in components like router (at the fig. 5). Router sends data to configurable of instances. Integrator in the configuration specifies number of elements in port array and their linkage with instances.

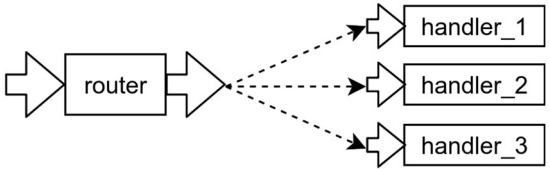

Fig. 5. Router has an array of out port which are connected to instances handler_1, handler_2 and handler_3

4.4 Memory blocks

Component instances in our system cannot use system heap, because there can be heap underflow with many instances and not enough heap size.

Access to heap and physical (for drivers) memory is done through ARINC-653 memory blocks. For each memory block component developer specifies:

- memory block name suffix

- memory size
- memory alignment
- flag, that shows if this memory block used by single instance or shared between instances.
- physical address for drivers working with memory mapped devices. Memory blocks with fixed physical address must be shared.

Name of shared memory blocks is identical to name suffix from configuration. Name of non-shared memory block is concatenation of instance name and memory block name suffix. Instances can access memory blocks by ARINC-653 API specifying memory block name.

4.5 Memory ownership

This part of the paper does not describe a feature of our approach. Here is some consideration on memory ownership.

Let us consider a component based system partition, which implement networking. There can be a track of components: Message_sender →UDP_IP_sender → Eth_sender→ Network_card_driver. Message sender sends pointer message to UDP_IP_sender; UDP_IP_sender prepends message with UPD and IP header and sends message to Eth_sender; Eth_sender prepends message with Ethernet header and sends to Network_card_driver. Should be specified how own memory and responsible for memory allocations.

If UDP_IP_sender and Eth_sender components would allocate buffers in their own memory, then this would greatly complicate their code, as they should also free buffers. Our real time C library does not support memory freeing because memory freeing can make indeterminate amount of time.

To simplify implementation and reduce overhead we used an approach when Message_sender allocates enough memory for all headers (component gets this value from configuration), copies message at the needed offset and pass to next layer pointer to message, message size, prepend and append values. Prepend describes how many bytes before message are allocated. Append describes how many bytes after message are allocated.

UDP_IP_sender to add header moves pointer it gets from Message_sender and decreases prepend value to header size.

5. Future work

We are going to work on supporting component distribution by binary images. This can be used to protect intellectual property of component developer, who does not want to share component source code.

Currently system integrator should specify component instances and their linkage in YAML language. We are going to support AADL language, which allows system integrator to graphically create and link instances. To work with AADL we are going

to use MASIW framework. MASIW [10, 11] (MASIW – Modular Avionics System Integrator Workplace) is s an open source Eclipse-based IDE for development and analysis of AADL models.

In addition, we are going to research possibility of using dataflow language to specify component, so that there will be no need to write component implementation in C language

6. Conclusion

In the paper, we presented a component-based approach that was created for JetOS, but can be used in other systems. The approach turned out to be efficient; it has low overhead and make system flexible and scalable while statically configured.

References

[1]. DO-178C, Software Considerations in Airborne Systems and Equipment Certification, January 5, 2012

[2]. Avionics application software standard interface part 1 – required services, ARINC specification 653P1-3, November 15, 2010

[3]. Mallachiev K.M., Pakulin N.V., Khoroshilov A.V. Design and architecture of real-time operating system. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 2, 2016, pp. 181- 192. DOI: 10.15514/ISPRAS-2016-28(2)-12

[4]. J. Siegel, Corba 3 fundamentals and programming, John Wiley & Sons, 2000

[5]. Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. 2001. Overview of the CORBA component model. In Component-based software engineering, George T. Heineman and William T. Councill (Eds.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA 557-571.

[6]. Distributed Component Object Model (DCOM) Remote Protocol Specification (online): https://msdn.microsoft.com/en-us/library/cc226801.aspx

[7]. Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. 2000. The SawMill multiserver approach. In Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system (EW 9). ACM, New York, NY, USA, 109-114. DOI: 10.1145/566726.566751

[8]. J. Liedtke. 1995. On micro-kernel construction. In Proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP '95), Michael B. Jones (Ed.). ACM, New York, NY, USA, 237-250. DOI: 10.1145/224056.224075

[9]. Boule I, Gien M, Guillemont M. CHORUS Distributed Operating Systems, Computing Systems, Vol. I No. 4 Fall 1988

[10]. D. V. Buzdalov, S. V. Zelenov, E. V. Kornykhin, A. K. Petrenko, A. V. Strakh, A. A. Ugnenko, and A. V. Khoroshilov. Tools for system design of integrated modular avionics. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 201–230 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-6

[11]. Alexey Khoroshilov, Dmitry Albitskiy, Igor Koverninskiy, Mikhail Olshanskiy, Alexander Petrenko, and Alexander Ugnenko. AADL-based toolset for IMA system design and integration. SAE Int. J. Aerosp., 5:294–299, 10 2012.

Использование модульного подхода во встраиваемых операционных системах

[1,2] *К.А. Маллачиев <mallachiev@ispras.ru>*

[1,2,3] *Н.В. Пакулин <npak@ispras.ru>*

[1,2,3,4] *А.В. Хорошилов <khoroshilov@ispras.ru>*

[1] *Д.В. Буздалов <buzdalov@ispras.ru>*

[1] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. Александра Солженицына, д. 25.

[2] *Московский государственный университет им. М.В. Ломоносова,*
119991, Россия, г. Москва, Ленинские горы, д. 1.

[3] *Московский физико-технический институт,*
141701, Россия, Московская обл., г. Долгопрудный, Институтский пер., д. 9.

[4] *Национальный исследовательский университет «Высшая школа экономики»*
101000, Россия, г. Москва, ул. Мясницкая, д. 20

Аннотация. Современные операционные системы для встроенных систем могут использоваться для решения задач управления в различных аппаратных контекстах. Управляющие ЭВМ могут различаться архитектурой центрального процессора, составом каналов связи, поддерживаемыми протоколами связи и т. д. Обычно встраиваемые ОС конфигурируются на этапе сборки, позволяя создать образ ОС, предназначенный для выполнения на определенной аппаратной платформе. Эту конфигурацию осуществляет команда, называемая группой системной интеграции. Зачастую ОС для встроенных систем разрабатываются множеством компаний. Если ОС не является модульной, то совместные проектирование, разработка и конфигурирование ОС представляют собой очень сложным задачи. Поддержка модульности и компонентой сборки значительно уменьшает необходимость во взаимодействии между компаниями-разработчиками. Клиентам это позволяет создавать минимальные решения, оптимально адаптированные под особенности задачи и аппаратной платформы. Кроме того, различные производители систем могут быть заинтересованы в том, чтобы внедрять в решение свои специализированные компоненты, в том числе и в бинарном виде, защищающем интеллектуальную собственность разработчиков. В данной статье мы представляем подход к построению модульных решений из гетерогенных компонентов на базе ОС РВ JetOS. Разработанный нами механизм связывания компонентов позволяет объединять гетерогенные компоненты от различных производителей в рамках одного раздела адресного пространства. Этот механизм позволяет разработчикам компонентов осуществлять независимую разработку. А системному интегратору позволяет независимо от разработчиков конфигурировать ОС, выбирая какие компоненты попадут в конечный образ ОС, и как эти компоненты будут взаимодействовать.

Ключевые слова: встраиваемые системы, модульность, компоненты, ОСРВ

DOI: 10.15514/ISPRAS-2017-29(4)-19

Для цитирования: Маллачиев К.А., Пакулин Н.В., Хорошилов А.В., Буздалов Д.В. Использование модульного подхода во встраиваемых операционных системах. Труды ИСП РАН, том 29, вып. 4, 2017 г., стр. 283-294 (на английском). DOI: 10.15514/ISPRAS-2017-29(4)-19

Список литературы

[1]. DO-178C, Software Considerations in Airborne Systems and Equipment Certification, January 5, 2012

[2]. Avionics application software standard interface part 1 – required services, ARINC specification 653P1-3, November 15, 2010

[3]. Mallachiev K.M., Pakulin N.V., Khoroshilov A.V. Design and architecture of real-time operating system. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 2, 2016, pp. 181- 192. DOI: 10.15514/ISPRAS-2016-28(2)-12

[4]. J. Siegel, Corba 3 fundamentals and programming, John Wiley & Sons, 2000

[5]. Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. 2001. Overview of the CORBA component model. In Component-based software engineering, George T. Heineman and William T. Councill (Eds.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA 557-571.

[6]. Distributed Component Object Model (DCOM) Remote Protocol Specification (online): https://msdn.microsoft.com/en-us/library/cc226801.aspx

[7]. Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. 2000. The SawMill multiserver approach. In Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system (EW 9). ACM, New York, NY, USA, 109-114. DOI: 10.1145/566726.566751

[8]. J. Liedtke. 1995. On micro-kernel construction. In Proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP '95), Michael B. Jones (Ed.). ACM, New York, NY, USA, 237-250. DOI: 10.1145/224056.224075

[9]. Boule I, Gien M, Guillemont M. CHORUS Distributed Operating Systems, Computing Systems, Vol. I No. 4 Fall 1988

[10]. Д.В. Буздалов, С.В. Зеленов, Е.В. Корныхин, А.К. Петренко, А.В. Страх, А.А. Угненко, А.В. Хорошилов. Инструментальные средства проектирования систем интегрированной модульной авионики. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 201-230. DOI: 10.15514/ISPRAS-2014-26(1)-6

[11]. Alexey Khoroshilov, Dmitry Albitskiy, Igor Koverninskiy, Mikhail Olshanskiy, Alexander Petrenko, and Alexander Ugnenko. AADL-based toolset for IMA system design and integration. SAE Int. J. Aerosp., 5:294–299, 10 2012.