社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2008－SLDM－134／2008－EMB－8 (17)
2008 / 3 / 27

A Realization of RPC in Embedded Component Systems

Takuya AZUMI†, Hiroshi OYAMA††, and Hiroaki TAKADA†

† Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan
†† OKUMA Corporation
5-25-1 Shimo-oguchi Oguchi-cho Niwa-gun, 480-0193, Japan
E-mail: ††{takuya,hiro}@ertl.jp, ††hi-ooyama@gmx.okuma.co.jp

Abstract This paper presents a Remote Procedure Call (RPC) channel in an embedded component system. The RPC channel is one of the components used in our component framework to facilitate the transparency of connected components in embedded systems. By supporting the RPC channel, it is possible to reuse the same component without modification on different communication mechanisms, including different task communications on the same or different processors and on networks. In addition, a memory allocator component and new keywords of a component description are provided. The effectiveness of our work is demonstrated.

Key words Component-Based Software Engineering, Embedded Systems, Real-Time Systems

1. Introduction

In recent years, the size and complexity of the software in embedded systems have increased. At the same time, the focus of software design in embedded system is moving away from a single processor towards a multiprocessor. To improve software productivity, software component technology for embedded systems has been increasing the attention from researchers [1]. Component technology for embedded systems, such as Koala component [2], PECT [3], PBO [4], and SaveCCT [5], have been developed so far. We have also developed an embedded component system, TOPPERS (*1) Embedded Component System (TECS [7], [8]), which is aimed to develop a base software for use in embedded systems. Features of TECS are the control-based connection, the static configuration, and the composite component. The control-based connection means that interconnection between components is done by the function call. The static configuration implies that both the configuration of component's behavior and the interconnections between the components are static. It is also possible to reduce the overhead of execution time.

This paper presents a new RPC Channel in the embedded component system. The RPC channel is one of the components used in TECS to facilitate the transparency of connected components. By supporting the RPC channel, it is possible to reuse the same component without modification on different communication mechanisms, including different task communications on the same or different processors and on networks. Moreover, developers are able to select a suitable RPC channel which depends on an environment when components are connected. Generally, RPC users are not able to select RPC channels because RPC channels of existing RPC technologies, such as CORBA [9], are hidden in the framework. In addition, a memory allocator component and new keywords of a component description are provided. The memory allocator component is useful for embedded systems because it is possible to select a suitable memory allocator in a variety of memory allocators. The keywords are provided to effectively use the memory allocators between components.

TECS is described in Section 2. Sections 3 and 4 represent the RPC channel for the Embedded component system. Summaries of this work are presented in Section 5.

(*1) : TOPPERS (Toyohashi OPen Platform for Embedded Real-time Systems) Project [6], which is based on the technical development results obtained by applying ITRON, is aimed to develop a base software for use in embedded systems.

2. Our Component System

In this section, the specifications of the TOPPERS Embedded Component System (TECS) are described.

2.1 Features of TECS

Embedded systems are usually considered to be resource-

constrained with respect to memory and must perform fast enough to fulfill their timing requirements. Typically, the greater the number of deadlines to be met is, the shorter the time between these deadlines needs to be. Thus, the component framework should influence the ability of the embedded system to accomplish its activities. Versatile component technologies, including JavaBeans and ActiveX for desktop applications, and the CORBA Component Model (CCM) and COM+ for distributed information, are generally unsuitable for embedded systems. For these technologies, the components are dynamically created and connected to other components during execution. This increases the overhead for creating, connecting, and calling a component. In the case of TECS, there is no need to reconfigure an application during execution. Consequently, to reduce overhead, components are statically created and connected to other components in the development of an application. This static configuration is the most important feature in TECS. As well, since TECS supports a variety of granularities of component, TECS can be used in different domains of an embedded system. A small granularity component, such as a device driver component, can be used to distinguish between the hardware-dependent and hardware-independent parts of the driver. A large granularity component, such as a TCP/IP protocol stack, is represented as a composite component consisting of two or more components. The benefit associated with using a composite component is the significant reduction in the complexity of component connection because several components can be treated as a single component, which enhances usability through the connection of components for an application developer (component user).

A variety of different types of components, such as memory allocators or RPC channels, are provided in order to increase reusability. For example, an RPC channel is one of the components used in TECS to facilitate the transparency of connected components. By supporting the RPC channel, it is possible to reuse the same component without modification on different communication mechanisms, including different task communications on the same or different processors and on networks.

2.2 Component Model

A *cell* is a component in TECS. *Cells* are connected properly in order to develop an appropriate application. A *cell* has *entry port* and *call port* interfaces. The *entry port* is an interface that provides services (functions) to other *cells*. The service of the *entry port* is called the *entry function*. The *call port* is an interface that uses the services of other *cells*. In this environment, a *cell* communicates with other *cells* using these interfaces. The *entry port* and the *call port* have *signatures* (sets of services). A *signature* is the definition of the interfaces in a *cell*. Interface abstraction using a *signature* provides control of the dependencies of each *cell*. The *cell type* is the definition of a *cell*, similar in scope to an object-oriented language *Class*. A *cell* is an entity of a *cell type*.

Figure 1 shows an example of the connection of *cells*. Each rectangle represents a *cell*. The left *cell* is an A *cell*, and the right *cell* is a B *cell*. Here, tA and tB represent the *cell type* name. The triangle in the B *cell* depicts an *entry port*. The connection of the *entry port* in the A *cell* describes a *call port*.

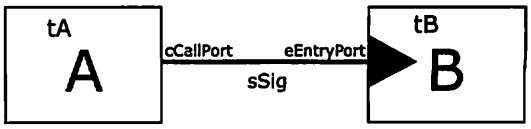

Figure 1 Example of Cells

A *call port* can only connect to an *entry port*. Therefore, in order to connect several *entry ports*, a *call port array* is used. An *entry port* can connect to *call ports*. However, in this case, it is impossible to identify which *call ports* are connected. A *singleton cell* is a particular *cell*, of which there is only one in a system. The *singleton cell* is used to reduce overhead because the *cell* can be optimized.

2.3 Component Description

A component description in TECS can be divided into three descriptions: a *signature* description, a *cell type* description, and a build description.

2.3.1 Signature description

The *signature* description is used to define a set of function heads. Figure 2 shows an example of *signature* description. A *signature* name, such as sSig, follows a *signature* keyword to define the *signature*. The initial letter of the *signature* name ("s") represents the *signature*. A set of function heads is enumerated in the body of *the signature* keyword.

```
signature sSig {
  ER func1(void);
  ER func2([in, size_is(size)] const char * buf,
           [in]          int size,
           [inout]       int *result);
  ...
}
```

Figure 2 Signature Description

The *in*, *out*, and *inout* keywords are used to distinguish whether a parameter is an input or an output. These keywords are understandable when a parameter is a pointer. In this case, the *result* parameter is used as input and output

because the previous result has an effect on the subsequent result. It is necessary to describe the size of an array by using *size_is* keyword in the TECS. The ER represents the error code of the return value.

2.3.2 Cell Type description

The *cell type* description is used to define the *entry ports*, *call ports*, *attributes*, and *variables* of a *cell type*. Figure 3 shows an example of *cell type* description. A *cell type* can have *entry ports*, *call ports*, *attributes* and *variables*., A *cell type* name, such as tA, follows a *celltype* keyword to define the *cell type*. The initial letter of the *cell type* name ("t") represents the *cell type*. To declare an *entry port*, an *entry* keyword is used. Two words follow an *entry* keyword: the *signature* name, such as sSig, and an *entry port* name, such as eEntryPort. The initial letter of the *entry port* name ("e") represents the *entry port*. Similarly, to declare a *call port*, a *call* keyword is used. The initial letter of the *call port* name ("c") represents the *call port*. The *attribute* and *variable* keywords that are used to increase the variety of *cells* are attached to *cell type* and are initialized when each *cell* is created. For example, a serial communication *cell* has an *attribute* to control the baud rate. Although each *attribute*, such as the baud rate, can not be changed during execution time, each *variable*, such as the file name of a file *cell*, can be changed. The set of *attribute* or *variable* keywords are enumerated in the body of this keyword. This keyword can be omitted when a *cell type* does not have an *attribute* and/or *variable*.

```
celltype tA {
  call sSig cCallPort;
};

celltype tB{
  entry sSig eEntryPort;
  attribute{
    ...
  };
  variable{
    ...
  }
};
```

Figure 3 Cell Type Description

2.3.3 Build description

The build description is used to declare *cells* and to connect between *cells* in order to create an application. Figure 4 shows an example of build description. To declare a *cell*, a *cell* keyword is used. Two words follow the *cell* keyword: a *cell type* name, such as tB, and the *cell* name, such as B. In this case, the eEntryPort (*entry port* name) of B (*cell* name) connects to the cCallPort (*call port* name) of A (*cell* name). The *signatures* of the *call port* and the *entry port* must be the same in order to connect *cells*.

```
cell tB B{
};
cell tA A {
 cCallPort = B.eEntryPort;
};
```

Figure 4 Build Description

2.4 Development Flow in the Framework

Figure 5 shows the development flow in the framework. The *signature* and the *cell type* descriptions are described by the component developers. The build description is written by the application developers. Based on the component descriptions, an interface generator can create C-language interface codes (.h or .c). Developers in this framework are divided into two groups: a component developer (component provider) and an application developer (component user). The role of the component developer is to define the *signatures* and *cell type* and to write the implementation codes (Component Source) for *cells*. Generally, a component is provided by the source code. On the other hand, the role of the application developer is to develop an appropriate application by connecting the *cells*.

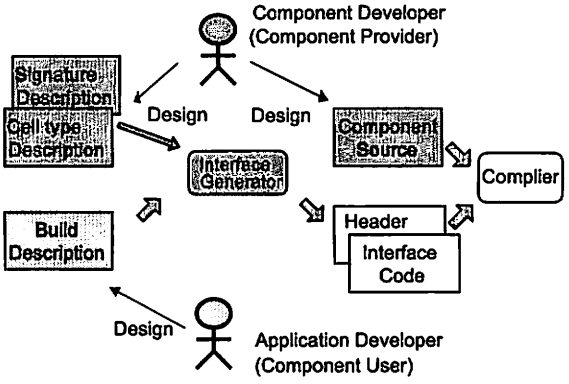

Figure 5 Development Flow in the Framework

3. RPC Channel in the Embedded Component System

3.1 RPC Channel Cell

An RPC channel is one of the components used in TECS to facilitate the transparency of connected components. By supporting the RPC channel, it is possible to reuse the same component without modification on different communication mechanisms, including different task communications on the

same or different processors and on networks. There are a lot of communication channels, such as a FIFO, a shared memory, a data queue, a mail box, TCP/IP, and so forth, in embedded systems. Therefore, the benefit of the RPC channel is that application developers are able to select a suitable RPC channel when the developers describe a build description.

Figure 6 represents an example of the communication between *cells* on different CPUs. In this case, cell10 (*cell* name) of tCT10 (*cell type* name) on CPU1 uses cell20 (*cell* name) of tCT20 (*cell type* name) on CPU2 through an RPC channel. The dual line between the *cells* shown in Figure 6 depicts an RPC channel. The RPC channel is one of the *cell types* in TECS. An RPC channel *cell type* is shown in Figure 7.

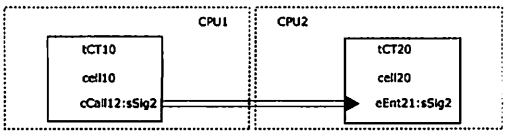

Figure 6 Connection of a Distributed Cell

The tRPCChannel is a *composite cell type* as shown in Figure 8. The *composite cell type* includes two or more *cells*. The benefit associated with using a *composite cell type* is the significant reduction in the complexity of component connection because several *cells* can be treated as a single *cell type*, which enhances usability through the connection of *cells* for an application developer (component user). The tRPCChannel can be divided into four parts: tServer, tMarshall, tUnmarshall, and tChannel *cell types*. The tServer *cell type* controls the RPC channel and calls functions of cell20. The tMarshall *cell type* is used to convert a standard date type for an RPC. The tUnmarshall *cell type* is used to reverse a standard data type. The tChannel *cell type* is used to transfer data that is converted by tMarshall.

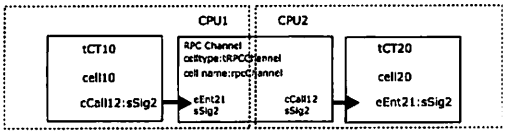

Figure 7 RPC Channel Cell Type

3.2 The through keyword

The *through* keyword is provided for inserting a *cell* between *cells*. This keyword is used to insert the RPC Channel into this framework. Figure 9 is an example of using the *through* keyword. The upper part of Figure 9 represents the component models, while the lower part of Figure 9 shows each build description. The *through* keyword is described just before the description (cCallPort=B.eEntryPort) of the

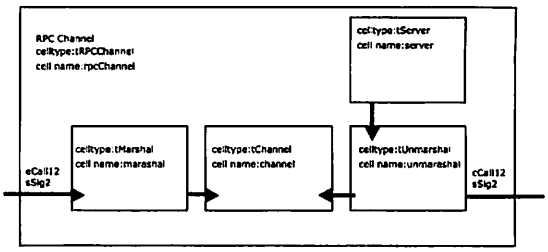

Figure 8 Composite Cell Type of an RPC Channel Cell Type

connection, which includes "Plugin" and "arg1". "Plugin" represents the type of plug-in. The plug-in is used to define the inserted *cells* and to generate C-language source codes when interface codes are generated by the interface generator. The application developers need to select the plug-in provided by component developers for inserting the *cell*. "arg1" gives additional information for the plug-in.

Although the *through* keyword is used to insert the RPC Channel, the keyword can be used for other purposes, such as inserting access control components [10]. An important advantage of using this description is that it is not necessary to modify the existing components. Therefore, it is possible to maintain the re-usability of components.


```
cell tB B{
};
cell tA A{
  cCallPort=B.eEntryPort;
};
```

```
cell tB B{
};
cell tA A{
  [through(Plugin, "arg1")]
  cCallPort=B.eEntryPort;
};
```

Figure 9 Example of Through

3.3 Memory Allocator

It is necessary to support a number of memory allocators in embedded systems to meet two requirements. The first requirement is that the developers should select a suitable memory allocator because there are memory allocators, such as a fixed-sized memory pool, a variable-sized memory pool and so forth. The second requirement is as follows. To avoid a memory fragmentation, a heap memory should be divided into several parts according to memory size. Also, according to task priority, the heap memory should be divided to avoid the case that high priority tasks can not allocate memory due to too much usage of a memory by low priority tasks.

In addition, there is a special purpose memory except the main memory, such as dual port memory on devices. In this case, specialized memory allocators are used to be allocated

from the special purpose memory. The required type or the number of allocators depends on applications. A memory allocator is one of the components used in TECS for application developers to freely select suitable memory allocators.

Figure 10 represents *signature* and *cell type* description of a memory allocator *cell*. The *signature* of memory allocator *cell* has two functions to allocate or deallocate memory. The *cells* which are connected the allocator *cell* can use the same memory allocator.

```
signature sAlloc {
  ER  alloc( [in]int32 size, [out]void **p );
  ER  dealloc( [in]void *p );
};
celltype tAlloc {
  entry sAlloc eA;
};
```

Figure 10 Allocator Component

3.4 Extended Signature Description

As mentioned in Section 2.3.1, TECS provides the *in*, *out*, and *inout* keywords to distinguish whether a parameter is an input or an output. The new keywords, *send* and *receive*, are provided to effectively allocate or deallocate memory in the component framework. These keywords are used for pointer parameters of functions. It is important to use these keywords with respect to memory allocation. A memory allocator for existing component system, such as Microsoft COM, do not need to pay attention about which an allocator is used because the same allocator is used in the systems.

The *in* and *send* keywords are used to transfer data from a caller *cell* to a calee *cell*. Figure 11 shows the difference between *in* and *send* keywords. In the case of using *in* keyword, a caller *cell* allocates and deallocates a data buffer. In the case of using *send* keyword, a caller *cell* allocates a data buffer and a callee *cell* deallocates. The *send* is used at the oneway situation which means a caller *cell* does not need to wait for finishing callee *cell*. Namely, the *send* is useful when a caller *cell* and a callee *cell* are executed in parallel.

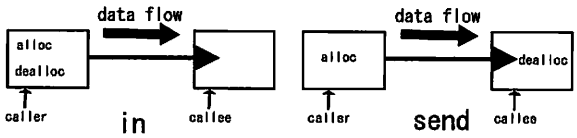

Figure 11 Difference Between *in* and *send* Keywords

The *out* and *receive* keywords are used to transfer data from a callee *cell* to a caleer *cell*. Figure 12 shows the difference between *out* and *send* keywords. In the case of using *out* keyword, a caller *cell* allocates and deallocates a data buffer. In the case of using *receive* keyword, a callee *cell* allocates a data buffer and a caller *cell* deallocates. The *receive* is useful when a caller *cell* gets the data which is unknown size in advance.

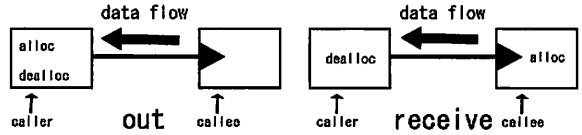

Figure 12 Difference Between *out* and *receive* Keywords

Two functions, which are used *send* and *receive*, are added into the *signature* description of Figure 2 as shown in Figure 13. Each of *send* and *receive* has a *signature* of an allocator *cell* like sAlloc in Figure 13. The *send* and *receive* are described just before a parameter (buf).

```
ER func3( [send(sAlloc),size_is(sz)]int8 *buf,
          [in]int32  sz );
ER func4( [receive(sAlloc),size_is(*sz)]int8 **buf,
          [out]int32  *sz );
```

Figure 13 Signature Description for *send* and *receive*

Figure 14 shows a build description for an allocator component. Allocator components connect to each parameter of *entry port* which is declared *send* or *receive*. It is not necessary to describe that allocator components connect *call port* because *cell* which has *call port* uses the same allocator of connected *entry port*. This description is described just before each *cell* declaration.

```
cell tAlloc alloc {
};
[allocator(
  eEntryPort.func3.buf=alloc.eA,
  eEntryPort.func4.buf=alloc.eA
)]
cell tB B{
};
```

Figure 14 Build Description for Allocator Component

4. Evaluation

The aim of this work is enable the development of an appropriate application by using an RPC channel. We implemented prototype RPC channels and evaluated the effectiveness of *send* on the RPC channels. Figure 15 shows the construction of the RPC channel which is used for evaluation. To

implement the RPC channel, date queues of ITRON-based real-time operating systems for single processor and multiprocessors are used. The RPC channels are generated by the interface generator. Function calls from *cell* A to *cell* B through the RPC are validated. This section compares the component programs with using *send* and *in* on the RPC.

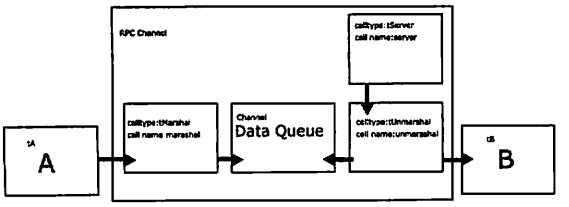

Figure 15 A Prototype RPC Channel

The number of clock cycle from *cell* A called to *cell* B completed is measured for each case. To measure the clock cycle, SkyEye [11] environment which simulates ARM architecture based microprocessors is used. In this experiment, this simulator is performed on a Centrino Duo 1.83GHz, running on Windows XP. The component programs and the RPC channel are performed on real-time operating systems which are TOPPERS/ASP kernel for a single processor and TOPPERS/FMP kernel for a multiprocessor. To implement allocator, the memory pool of these RTOS is used. Table 1 shows the number of clock cycle for *in* or *send* on the RPC of a single processor and a multiprocessor. The numbers in Table 1 are the average of clock cycle for 100 iterations. The result implies that *send* is greater than *in* at the oneway situation for both RPC channels. The oneway situation means a caller *cell* does not need to wait for finishing callee *cell*. The difference of the clock cycle is because the input data is copied for executing in parallel in the case of using *in*. Therefore, *send* improves the performance compared with *in* at oneway situation. Namely, *send* is useful at parallel execution using the RPC.

Table 1 Comparison of The Numbers of Clock Cycle

	RPC(single)	RPC(multi)
in (oneway)	22,124	20,063
send (oneway)	20,355	18,371

5. Conclusion

This paper describes a Remote Procedure Call (RPC) channel in an embedded component system. The RPC channel is one of the components used in our component framework to facilitate the transparency of connected components in embedded systems. By supporting the RPC channel, it is possible to reuse the same component without modification on different communication mechanisms. In addition, a memory allocator component, and *send* and *receive* keywords are provided. The memory allocator component is useful for embedded systems because it is possible to select a suitable memory allocator. The keywords are provided to effectively use the memory allocator between components. Moreover we implemented prototypes of RPC channels and demonstrated the effectiveness of *send* keyword on the RPC channels.

Acknowledgment

The authors would like to thank the TOPPERS component working group for their helpful comments and suggestions.

References

[1] I.L. Yen, J. Goluguri, F. Bastani, and L. Khan, "A component-based approach for embedded software development," Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002. (ISORC 2002), Washington, DC, USA, pp.402–410, 29 April-1 May 2002.

[2] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The koala component model for consumer electronics software," IEEE Computer, vol.33, no.3, pp.78–85, March 2000.

[3] K.C. Wallnau, "Volume iii: A component technology for predictable assembly from certifiable components.," Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburg, USA, 2003.

[4] D.B. Stewart, R.A. Volpe, and P.K. Khosla, "Design of dynamically reconfigurable real-time software usingport-based objects," Software Engineering, vol.23, no.12, pp.759–776, December 1997.

[5] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli, "The save approach to component-based development of vehicular systems," Journal of Systems and Software, vol.80, no.5, pp.655–667, May 2007.

[6] TOPPERS Project. http://www.toppers.jp/en/index.html.

[7] T. Azumi, M. Yamamoto, Y. Kominami, N. Takagi, H. Oyama, and H. Takada, "A new specification of software components for embedded systems," Proceedings 10th IEEE International Symposium on Object/component/service-Oriented Real-Time Distributed Computing, 2007. (ISORC 2007), Santorini Island, Greece, pp.46–50, 7-9 May 2007.

[8] T. Azumi, S. Yamada, H. Oyama, Y. Nakamoto, and H. Takada, "Visual modeling environment for embedded component systems," Proceedings IEEE 7th International Conference on Computer and Information Technology (CIT 2007), Fukushima, Japan, 16-19 October 2007.

[9] OMG, "CORBA." http://www.omg.org/corba/.

[10] T. Azumi, S. Yamada, H. Oyama, Y. Nakamoto, and H. Takada, "A new security framework for embedded component systems," the 11th IASTED International Conference on Software Engineering and Applications, Cambridge, Massachusetts, USA, pp.584–589, 19-21 November 2007.

[11] SkyEye. http://www.skyeye.org/index.shtml.