Interfacing Remote Software Components using Dynamic Token Code

Dr.-Ing. Egmont Woitzel
University of Rostock, Department of Electrical Engineering,
Institute of Applied Microelectronics and Computer Science

Abstract: *The development of distributed controller applications demands flexible and efficient interface technologies. Following the paradigm of „component software" in current desktop programming practice, distributed applications are considered to consist of communicating components. Furthermore, open interfaces are necessary for interaction between debuggee and development tools during integration and test of distributed components.*

Unfortunately, universal interface technologies like OLE or OpenDoc does not meet the requirements of small embedded systems because of their huge resource consumption.

Nowadays, low-level interpreting techniques are frequently used to implement open interfaces. OpenBoot, Java and SENDIT are examples for that approach. All of them use more or less fixed instruction sets for controlling their target systems.

In this paper an open interface technique, which uses dynamic token code for parameter transfer and function calling, will be described. The tokens used for function calls will be determined at initialisation-time when predictable run-time behaviour is required or otherwise during the run-time for debugging purposes. The process of dynamic binding of function names to tokens may be completely transparent to the calling application.

A major advantage of the dynamic approach is that application components may be separately maintained without the need of updating the calling components. Furthermore, the dynamic token binding allows to implement interactive debugging environments.

1 Component Software Development

Recent trends in computing industries let expect that reusability of software in future will be transported by means of component software rather than by means of object-oriented programming languages. Object models like CORBA or COM define powerful universal interface techniques which include not only mechanisms of function calling and parameter transfer but also the process of finding and instantiation of objects, e. g. components. This mightiness tends to resource consuming implementations which does not meet the limitations of small and cost sensitive embedded systems.

Nevertheless, distributed embedded software systems perfectly fit to the idea of interacting components. First of all, the data encapsulation provided by strong defined functional interfaces will increase the software quality. Depending on the component granularity the process of software development will be determined by the process of debugging component interfaces rather than of debugging procedures. Consequently, tailored interface techniques will be also required for debugging purposes.

In contrast to common desktop or office applications components of distributed embedded applications most often depend on specialised hardware like process interfaces. Therefore, most software tasks were bound to a special network node. Localisation of software components and dynamic instantiation will be of less importance, especially in the case of small applications. On the other hand, due to the heterogenous character of typical controller networks interface technologies have to be machine independent.

2 Embedded Interpreters and Virtual Machines

In the following we will use the term client for component which uses interface functions or service of an other component. This component will be called server component. There are at least two general methods to provide a machine independent interface by a server component:

- embedded interpreters
- virtual machines

When using the embedded interpreter method, the server component accepts programming language like command strings from the client, which will be interpreted at run-time. This approach is known from languages like PostScript[1] [1]. Other interpreted languages like Forth [2] will be often used as control language in process automation. Rohrer, Schnitter and Walchshäusl describe a Forth based network application which controls a linear accelerator [5]. Typically, the interpretation of command strings causes run-time overhead and consumes memory resources. Sun Microsystems Corporation uses a refined approach to provide CPU independent driver code for workstations, called OpenBoot [7]. A so-called tokenizer translates Forth source code to FCode. FCode consists of tokens which are numerical representations of keywords. Instead of parsing source code searching strings the FCode interpreter performs simple table lookup. Nevertheless, user defined symbols have to be searched as strings in a symbol table.

Another way to implement a machine independent server interface is to implement a virtual machine. A virtual machine will be represented by a well-defined set of instructions. In the context of component software the instruction set may be considered as the set of services provided by a server component. Virtual machines are in widespread use for implementing machine-independent language run-time-systems. Best known example for that approach is the Java virtual machine definition [6], which allows network-wide distribution of executable code.

3 Dynamic Token Code

The most important drawback of the decribed interface technology consists in the fixed binding between token representation and the service that should be executed. Especially in the case of devolopment and debugging scenarios the meaning of a token may be often subject of change. A more loosely kind of binding could be achieved, if character strings were used to label a service. The benefits of fast execution behavior and portability of token code will be preserved, if we assign tokens to name strings at application run-time.

4 Building Transparent Interfaces Using Dynamic Token Code

A transparent and open interface technology based on dynamic token code has to cover following problems, which will be discussed below:

- session management
- creation and execution of token lists
- managing data exchange
- managing token resolution

At first, a bi-directional conversation has to be established between client and server. The client will send service requests and input parameters to the server and the server will respond with resulting data. The underlying communication system has to secure the correctness of all transferred data and to handle network routing and related tasks. The lifetime of such a conversation will be called session. A session will be usually requested and closed by the client component. The termination of a session by the server will be considered as an exception. The validity of tokens will end with the termination of a session. Therefore, a server component may be replaced without any side effects if there is no active session.

After opening of a session the client application may request the execution of services by issuing tokens. We will see later that token binding and data exchange may be reduced to token issuing. To reducing the protocol overhead of the underlying communication system it is possible to defer the physical transmission of submitted requests until one of the following conditions occur:

- the next request overflows the transfer buffer size
- the application asks for server response
- the application explicitly synchronises the server

When the server component receives such a token list, it simply executes the corresponding services. Because of the server local coding scheme this process may be implemented very efficient. If an indirect threaded code technique will be used, a service call takes about twice the cost of a machine level call. In that case tokens have the size of addresses and will be interpreted as pointers.

This simple view of token list has to be extended for parameter transfer. Calling remote services without any parameters makes little sense in most applications. Universal interface technologies like RPC or OLE use expensive methods at client and server side to translate and transfer parameters. Because of its restricted resources it is not possible to implement such powerful techniques inside of small embedded systems. In the context of a highly distributed application most of these small systems will host server components, like sensor interfaces or local controllers. Therefore, we have to optimise the parameter transfer

protocol for minimal resource consumption on the server side. One simple solution consists in separating the parameter transfer from the service call. Parameters will be transferred to a server-side data structure and services will take their inputs from and put their outputs to that structure. Best performance will be achieved, if a stack is used for that purpose. Furthermore, this kind of interface matches the architecture of virtual machines like Java and OpenBoot as well as the programming model of languages like Forth or Postscript.

The following table shows a C-based programming interface described above.

Function	Description
Session Management	
`CID ClientInitialize()`	connect client to interface manager
`void ClientUninitialize(CID)`	disconnect from interface manager
`TID ClientConnect(CID, char far *profile, char far *section)`	connect client to server component based on profile information
`void ClientDisconnect(CID, TID)`	disconnect server component
Service Management	
`SVID GetService(CID, TID, char far*)`	resolve service token
`void DoService(CID, TID, SVID)`	execute service token
`void FlushServices(CID, TID)`	flush token list buffer
Parameter Transfer	
`void PushInteger(CID, TID, INT)`	transfer integer to server stack
`void PushCharacter(CID, TID, CHAR)`	transfer character to server stack
`void PushLong(CID, TID, LONG)`	transfer long integeer to server stack
`void PushString(CID, TID, char far*)`	transfer string to server stack
`INT PopInteger(CID, TID)`	retrieve integer from server stack
`CHAR PopCharacter(CID, TID)`	retrieve character from server stack
`LONG PopLong(CID, TID)`	retrieve long integer from server stack
`UINT PopString(CID, TID, char far*, UINT)`	retrieve string from server stack

5 Binding Strategies

There are two different client strategies to bind tokens to services. Applications with a fixed and well-known number of services may bind all token at *start-up-time*. That allows simple compatibility checking and avoids run-time failures caused by non-implemented services. Furthermore, there is only a very small and deterministic overhead at run-time to send a service request.

On the other hand, applications may bind tokens on *demand*. This strategy is useful for implementing applications with a large number of possible but seldom called service requests. In this case every first service request will be preceded by a corresponding token resolution request. One advantage of this approach is a shorter start-up-time of an application. This dramatically reduces the necessary bandwidth of the communication network in exceptional situations like the simultaneous start-up of all components after a system-global power-on. We will see that generic applications like interface explorers, object inspectors or other interactive programming tools will benefit from this strategy, too.

6 Enhancements

Further refinements are possible. In the case of very small server component host systems the binding service may be offered by a separate name server. The fieldFORTH programming environment [8] uses that strategy.

Some programming systems like Forth define procedures for generating executable code. In that case it is possible to map program source code translation actions to service request, which enables application programs to define target resident executable code. This feature may be exploited to optimise network traffic and to balance the workload between network nodes.

7 Generic Development Tools

The bind-on-demand-strategy allows to implement generic interactive development tools. There is no need to compile service name strings into the development application for command-line type tools. After typing in service name strings the tool resident command line interpreter can bind the corresponding token and send it to the server component. Cache techniques should be used for better response behavior.

Nevertheless, the implementation of a small number specialised interfaces comparable to Microsoft's OLE automation allows to implement some more advanced debugging tools, such as interface exploration tools and status inspecting tools.

8 Planned Extensions

Dynamic token code interfaces have been used in different project, including programming environments like fieldFORTH, a generic DDE-Interface for the Forth programming languages and an application interface between a PC-hosted image processing system and a DSP TMS320C40 camera and preprocessing system [3].

Current development targets a generic gateway between dynamic token code interfaced server components and Microsoft ActiveX control technology.

9 References

[1] Adobe Systems Corporation: "Postscript Language Reference Manual".

[2] American National Standards Institute, Incorporated: American National Standard for Information Systems - Programing Languages - Forth. Document ANSI X3.215-1994

[3] Höhenleitner, Thomas: EFA's RIB. Proceedings of the EuroFORTH '95 Conference, Dagstuhl Castle 1995

[4] Microsoft Corporation: „RPC Programmer's Guide and Reference“, Microsoft Developer Network, Library April 1996

[5] Rohrer, L.; Schnitter, H.; Walchshäusl, B.: ONF Open Network Forth. Proceedings Forth '93. Forth-Gesellschaft e.V., Nürnberg 1993

[6] Sun Microsystems Computer Corporation: The Java Virtual Machine Specification. 1995, Online Document

[7] Sun Microsystems Computer Corporation: OpenBoot Command Reference. Manual Revision A, October 1993

[8] Woitzel, Egmont: Emulating Forth: Interactive Cross Development. Proceedings of EuroFORTH '95, Dagstuhl Castle 1995

[1] PostScript is a rgistered trademark of Adobe Systems Inc.