Portable Data Exchange for Remote-Testing Frameworks *

Raimund Kirner, Peter Puschner, Ingomar Wenzel, and Bernhard Rieder
Institut für Technische Informatik, Technische Universität Wien
Treitlstraße 3/182/1, 1040 Wien, Austria
{raimund,peter,ingo,bernhard}@vmars.tuwien.ac.at

Abstract

To communicate between heterogeneous computer systems, mechanisms for data conversion are necessary. In this paper we present a portable, asymmetric data conversion method that is suitable for remote testing frameworks in embedded systems development. The described method takes the resource limitations of embedded systems into account by doing the data conversion at the testing host. The method can be implemented as platform-independent source code and it avoids the need of recompiling the code of a communication partner if the code of the other communication partner is migrated to a different platform.

1 Introduction

Embedded systems often have limited resources in order to achieve low power consumption and competitive production costs. Due to these resource limitations, testing embedded systems is typically done as *remote testing*, i.e., the local (host) system communicates with components that run on the remote (target) system. Remote testing is often needed since an embedded system hardly has enough free resources to generate the test data and store the test results locally.

As there are many different microcontrollers used in embedded systems, a testing framework for remote testing has to be easy to adapt to new target platforms. However, the potentially different data representation of the host and the target system are a challenge when designing a portable testing framework.

In general, the problem of exchanging data between heterogeneous computing platforms is not new and several proper solutions for converting the data formats between communication partner exist. The development of these protocols was motivated by the need to write software that runs on heterogeneous computing platforms and by the need of communication between software components that run on the different nodes of such computing platforms.

Three mechanisms are required for data exchange between heterogeneous computing platforms: a) the (de)serialization of data structures, b) the conversion between different data formats, and c) a low-level data communication channel. Within this paper we discuss a novel resource-aware realization for a) and b).

The conversion of the data format between a sender and a receiver can be done in two ways: by *asymmetric* or *symmetric conversion*.

In the *asymmetric conversion* the data conversion is done only at one designated end of the communication channel, i.e., at the sender or at the receiver side. The converter has to know information about the data representation of the platform at the other end of the communication channel. The Network Data Representation (NDR) defined by the Distributed Computing Environment (DCE) standard [10] solves this problem by embedding the sender description to the communicated data. Their approach is called "receiver-makes-right" as the conversion is always done at the receiver side. NDR is used, for example, in Microsoft COM/DCOM [1] or CORBA [5]. As shown in Figure 1, the concept of "receiver-makes-right" requires a receive converter at both communication partners. *Conv* and *Conv'* denote the data conversion for sending/receiving data and *Ser* and *Ser'* denote the serialization/deserialization of data. *Snd_B* and *Rcv_B* mark the low-level data communication channel.

A *symmetric conversion* method uses a *canonical* intermediate data format for data exchange. The advantage of the symmetric conversion is that none of the communication partners needs to know the data representation of the partner located at the other end of the communication channel. The disadvantage of sym-

*This work has been supported by the FIT-IT research project "Model-Based Development of distributed Embedded Control Systems (MoDECS)" and the ARTIST2 Network of Excellence of IST FP6.

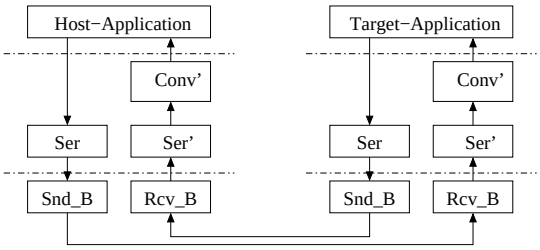


Figure 1. "Receiver-Makes-Right" Method

metric conversion is that it requires converters at both ends of the communication channel, i.e., a conversion of data into the intermediate data format is always done, even if the sender and the receiver use the same data representation. Examples for symmetric conversion are Sun RPC, which is based on the XDR (external data representation) [8], or Jini [9], which uses the Java Format [4]. The concept of symmetric conversion is shown in Figure 2. This concept requires two converters (one for coding and one for decoding) at both ends of the communication channel.

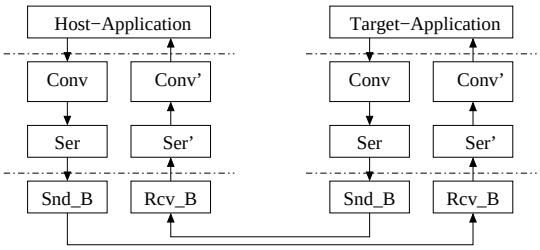


Figure 2. "Symmetric Conversion" Method

The existing approaches to data conversion for communication between heterogeneous computing platforms have not been designed to fulfill the special needs for remotely testing embedded systems. In the latter, resources on the target system are often scarce and the data-conversion overhead on the target should be minimal. For example, the code memory of embedded computers might be almost occupied by the application code but it may be required to test all software components at once. As shown above, existing data exchange techniques require the allocation of memory and computation resources for at least one converter on the embedded system. Furthermore, the data exchange format of the above techniques are not compact, which could become significant when exchanging a lot of data over a slow communication channel.

In this paper we describe a novel, portable data exchange method that is well-suited for the remote testing of embedded systems. The data conversion is asymmetric and always done at the host side ("host-makes-right"). Note that this approach has a significant benefit over the existing techniques of *symmetric conversion* or "sender/receiver-makes-right". With the "host-makes-right" approach, the target system under test only needs few memory and computation resources for data exchange. It does not need a data converter. The principle of the data conversion method and its principal integration into testing frameworks is described in Section 2. Section 3 presents a timing analysis framework that uses our data exchange method. Additional related work of remote testing frameworks is discussed in Section 4.

2 Portable Data Exchange with a System having Restricted Resources

This section describes the principle of our data exchange method between host and target and its integration into testing frameworks. The data exchange method uses an *asymmetric conversion* that is always done on the host side. To do so, the host uses a platform description about the target. This platform description has to be calculated only once. We assume that a communication layer is available that can transmit streams of bytes. For the efficient implementation of the method we assume a programming language that provides pointers and static type casting (for example, *C* or *C++*).

2.1 Data Exchange

We are interested in the bidirectional communication between a *host* computer without relevant resource limitations and a *target* computer with strict resource limitations. The target computer may have significant limitations on both, computation and memory resources. Furthermore, the bandwidth of the communication medium available for testing may be also considered as a significant bottleneck.

Beside being able to deal with the resource limitations, our data exchange method should be implementable as highly portable source code. By high portability we understand software that runs on common platforms without the need of modifications. Future platforms may still require an adaption of the implementation. Under platform we understand both, software and hardware. The software platform basically consists of the compiler and the runtime environment.

As shown in Figure 3, our "host-makes-right" approach uses *asymmetric conversion* of data to keep the resource requirements at the *target* side low. Hence, the conversion will be done at the sender side for host

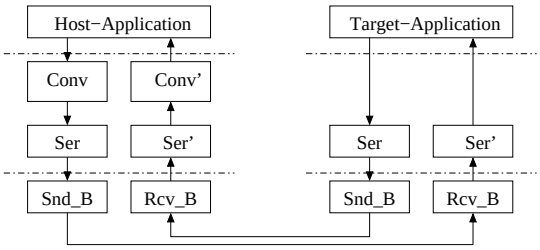


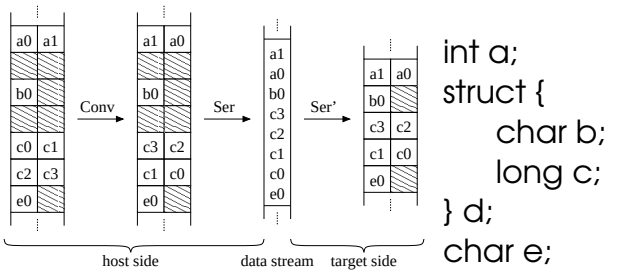
Figure 3. "Host-Makes-Right" Method

to target communication, or at the receiver side for the transmission from the target to the host.

This communication scenario is realized in two steps:

1. **Determination of Target Platform:** Prior to any regular data exchange, the target computer calculates and sends information that describes the data representation at the target side (Inf_T) to the host computer. Inf_T has all its data in an appropriate order so that it can be directly interpreted by the host computer. To keep resource usage at the target low, Inf_T can be calculated before loading the actual application program to the target.

2. **Regular Data Exchange:** Having received Inf_T from the target, the host can parameterize its data converters. It is then ready for the regular data exchange with the target computer.



(a) Data Flow from Host to Target (b) Data types to exchange

Figure 4. Example of Data Exchange: Sending Data from Host to Target

Figure 4 provides a simple example of how our data exchange method works. It demonstrates how to resolve different endianness and memory alignment between host and target. We assume that the data given in Figure 4(b) have to be transmitted from the host to the target. The data flow of this data exchange is given in Figure 4(a). The left-most memory dump shows the original placement of the data in the memory of the host. Hatched areas mark unused memory due to the concrete alignment of the data types. We assume that the byte order is *little endian* on the host and *big endian* on the target, for which the conversion is done by *Conv*. The host serializes the basic data types into a stream of bytes, which it then sends over the communication channel to the target. The target reads the data and de-serializes them into its own data alignment. In practice, often more complicated conversions than shown in the above example can be required. But again, all conversion steps are done by the host. The specific benefits of this data exchange method are:

- There is no need for a data-representation converter on the target system; thus memory and computation resources can be saved;
- A communication layer on top of the data exchange mechanism does not need to transmit information about data representation of the message.
- The data exchange mechanism can be implemented as highly portable source code.

Note, that the data exchange method is independent from the communication protocol used. It only requires that the communication protocol supports the transmission of sequences of bytes. As the choice of the communication protocol is application-specific, it cannot be part of our portable data exchange method.

There are two types of application scenarios of our data exchange method:

Testing Tool: This was the application we had in mind when we developed this data exchange method. In this case Inf_T is also used by the host to adapt the test case generation to the value range of the data types of the target computer.

Distributed Application: it is also possible to use our approach for the development of a portable distributed application, but it leaves the problem of dealing with the fact that data types may cover different value ranges when used on different platforms in the responsibility of the developer, e.g., by using run-time checks.

The following describes details and applications of the "host-makes-right" data exchange method.

2.2 Description of Target Platform

The description of the data representation (Inf_T) at the target platform is needed by the host computer prior to any regular communication.

Inf_T is calculated automatically by running a program dedicated for this purpose at the target computer. However, the code to calculate Inf_T is typically more complex than a reasonable implementation of a data-representation converter that converts only between the local data representation and one external data representation. Hence, to get an improvement in resource consumption by our data exchange method, the code to calculate Inf_T has to be executed on the target before the regular application is loaded and executed (once Inf_T has been computed, this code is no longer needed and can be removed from the target).

When using embedded systems where nodes are equipped with a boot loader, the prior execution of the routine to calculate Inf_T can be directly integrated into the startup process of the system. Alternatively the host may store Inf_T for a target in a configuration file. In that case Inf_T need not be re-calculated again and again, but may be read from this file when it is needed. The latter approach fits better to systems where uploading the code is not that easy as in case of a boot loader.

Listing 1. Calculation of Platform Properties

```
1 unsigned int var = 1;
2 endian = ( *((char *)(&var)) ==
3     (char)(var) ) ? LITTLE : BIG;
4 len_sc = sizeof(char);
5 len_uc = sizeof(unsigned char);
6 len_ss = sizeof(short);
7         ...
```

Listing 1 should give an idea of how the target description is calculated for integral data types, as only their size together with the byte order (endianness) is needed to describe them. Note that for simplicity of the presentation, the shown byte order test is quite naive as it assumes that the platform has either *little* or *big* endianness, while exotic platforms may also have a different byte order (called *mixed endianness*). The calculation of floating point properties is a little bit more complex because there exists a broader variety of different implementations. An algorithm to calculate properties of floating point arithmetic has been published by Cody [2]. A detailed description of how to calculate all platform properties relevant for our data exchange technique is omitted here, but can be found in an article of Pemperton [7].

2.3 Serialization of Data

Before data structures can be sent over a network they have to be *serialized* [1] into a flat stream of bytes. At the receiver side the data will be *deserialized* into their new local shape. Network data representations such as NDR [10] or XDR [8] provide interface definition languages (IDL) that can express also complex data structures. For IDL there exist tools to generate the required stub code at the sender and the receiver to (de)serialize the data from the specification.

For our data exchange method we do (de)serialization in a similar way. Data structures are transmitted as a sequence of their elements of basic data types. As in ANSI C, arrays are considered as basic data types since they resemble a flat byte stream. The (de)serialization is done by some extra code which we call *glue code*. For example, the glue code is responsible to copy the content of elements of basic data types. On the other hand, the glue code does not have to consider the specific memory layout of data structures as this is resolved by the compiler.

Although the (de)serialization process of our data exchange method is conceptionally similar to other approaches, the new concept is that we focus on minimizing the overhead required at the target side and reducing the required bandwidth over the communication channel. For example, the data to be communicated after serialization are represented as a sequence of bytes without overhead.

In the following we describe a possible implementation strategy for the glue code at the target side for data exchange. The implementations tend to be quite short and therefore are also suitable for manual coding. Of course, tool support to generate them automatically is preferred. The implementation for the data exchange at the host side is more complex as it also has to contain the converter to adapt the data representation.

One implementation strategy uses a *serialization table* at the target to copy elements of basic data types after receiving or before sending data. The *serialization table* holds an entry for each element of a basic data type to be communicated. Each entry consists of a size field that denotes the size in bytes of the data element, and of a reference to the memory location of that element. The serialization table has the following type definition in ANSI C:

```
typedef struct
{ void *ref; int size; } data_list_t;
```

To give an example of how the serialization table can be constructed, we assume that we want to receive

[1] serializing is also referred to as *marshaling*

data values for the variables given in Listing 2. The resulting glue code to initialize the serialization table is depicted in Listing 3.

Listing 2. Example: Variables to Exchange

```
typedef struct { int a[3]; char b; }
  Sn_t;
char cv;
int  anv[100];
Sn_t sv;
```

Listing 3. Example: Serialization Table

```
data_list_t  sertab[] = {
  { &cv,     sizeof(cv)   },
  { &anv,    sizeof(anv)  },
  { &(sv.a), sizeof(sv.a) },
  { &(sv.b), sizeof(sv.b) }
};
```

To illustrate how (de)serialization is done, we show the simple glue code in ANSI C for the deserialization of data in Listing 4. It uses a routine **char** get_byte() to read from the received byte stream. The data elements are copied byte-wise via the serialization table to the local memory locations. The code for serialization is similar and therefore omitted.

Listing 4. Deserialization at the Target using the Serialization Table

```
void deserialize (data_list_t dl[],
                  int cnt)
{ int i,j;
  for (i=0; i<cnt; i++)
  {
    for (j=0; j < dl[i].size; j++)
    {
      ((char*)(dl[i].ref))[j] =
        get_byte();
    }
}
```

2.3.1 Optimizing Code Size of Serialization

The above glue code based on a serialization table is relatively compact when communicating a certain amount of data, especially in case of arrays. The more data have to be exchanged, the more effective is this solution. However, if the data to be communicated are small, it is more efficient to use glue code working with direct data assignments instead of using a table together with auxiliary routines for (de)serialization. In case the glue code is generated by a tool, this tool may also determine from the concrete application code which glue code implementation strategy results in a smaller memory footprint at the target computer.

2.4 Integration in Remote Testing Framework

Even though the described data exchange method is generally applicable, our main concern is its use for the remote testing of embedded systems. For remote testing the (de)serialization is the same as described above. But the glue code may need additional variable declarations or data assignments to interface with a specific software routine to be tested.

For example, for each input parameter of the software under test it is necessary to allocate a variable within the glue code where the actual parameter value can be stored. This is required independently of whether the parameter is passed by value or by reference. The same approach is required if an input parameter forms dynamic data structures. In this case the glue code has to allocate several data structures that are linked together by pointer references.

Furthermore, testing may also require additional monitoring data to be reported to the host computer. As above, these monitoring data also have to be copied by the glue code.

The code that emits the monitoring data for testing purposes is something specific to a concrete testing technique and not part of the glue code used to interface the software routine with the communication line.

2.5 Broadcast Messages

The data exchange method described in this paper is designed for point-to-point data exchange. There is only a limited support for broadcast messages. The reason is that the host as sender already prepares data to suit the format of the receiver. But this does not work if multiple receivers with different data representations should be addressed simultaneously. A possible solution is to use broadcast messages carrying 4-bit integer values. These 4-bit values can be symmetrically encoded twice within a 8-bit byte so that endianness of the receiver does not matter.

3 Application Scenario

The data conversion method described in this paper was developed to increase the portability of remote testing frameworks used to test embedded systems. Our application domain of interest is the determination of the worst-case execution time (WCET)

of embedded systems. Concretely, we have developed a measurement-based WCET analysis framework that combines static and dynamic methods [3, 11]:

- Static code analysis to ensure that all feasible paths of the program are taken into account.
- Execution-time measurements to avoid the burden of modeling the instruction timing of the target processor.

For the execution-time measurement we have to send test data to the target computer (embedded system) and receive the corresponding measurement results. This WCET analysis framework is a typical application domain for the "host-makes-right" data exchange method described in this paper, because the WCET analysis framework has to be portable to different target platforms. A further requirement was that the framework uses only few resources at the target computer so that it is possible to test the complete software running on a node.

To give an example of the memory overhead of our data exchange method, lets assume again that we want to exchange the data given in Listing 2. Using the serialization table approach, the additional memory consumed by our data exchange method on an 8-bit AVR microcontroller as target system was `186 byte` for code and `14 byte` for the table. To compare, on a Cygwin Pentium system it needed `111 byte` for code and `28` byte for the table. Thus, this data exchange method demands quite few resources from the target system.

4 Related Work

The conversion of the data representation to communicate between heterogeneous platforms has been already tackled in various ways. A discussion of existing *symmetric* and *asymmetric* conversion methods has been already given in Section 1.

Regarding the application of data exchange techniques, there exist remote testing frameworks that already address the issue of portability in a certain extend. TETware [6] from the Open Group is such an example. The approach done in TETware is to exclude the data conversion from the core framework. Instead, it provides only an interface definition for a communication subsystem. The users of the testing framework have to implement their own communication subsystems that perform the data conversion in case of heterogeneous systems. Our approach reduces the effort needed to port a remote testing framework to a new platform. By including the data conversion into the framework we reduce the effort required for implementing a new communication subsystem for a specific platform.

5 Summary and Conclusion

Embedded systems typically have only limited resources. The resource limitations are becoming even more challenging when additional resources are needed to remotely test the embedded system.

In this paper we have described a bidirectional data exchange method for heterogeneous systems. This data exchange method has been designed to be used within remote testing frameworks for embedded systems.

This data exchange technique needs only few resources at the embedded node because data conversion is done in both directions at the host computer.

The efficiency of this data exchange technique is high, because it does not need to send additional meta information describing the data format of the sent data.

The data exchange technique is portable because it can be realized as portable source code. We have also discussed how the platform properties of the embedded system that are relevant for data representation can be calculated by software.

A measurement-based worst-case execution time analysis framework has been described as a possible application scenario of this data exchange method.

References

[1] N. Brown and C. Kindel. *Distributed Component Object Model Protocol – DCOM/1.0.* Network Working Group, Jan. 1998. Internet draft.

[2] W. J. Cody. Algorithm 665, MACHAR: A subroutine to dynamically determine machine parameters. *ACM Transactions on Mathematical Software*, 14(4):303–311, Dec. 1988.

[3] R. Kirner, P. Puschner, and I. Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. In *Proc. 4th International Workshop on Worst-Case Execution Time Analysis*, pages 67–70, Catania, Italy, June 2004.

[4] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition).* Addison-Wesley, 1999. ISBN 0-201-43294-3.

[5] Object Management Group. *Common Object Request Broker Architecture: Core Specification,*, Dez. 2002. Version 3.0.

[6] The Open Group, Berkshire, UK. *TETware - Realtime and Embedded Systems Extension*, Jun. 2003.

[7] S. Pemperton. The ergonomics of software porting: Automatically configuring software to the runtime environment. Technical Report CS-R9266, Centrum

voor Wiskunde en Informatica (CWI), Amsterdam, Dec. 1992.

[8] R. Srinivasan. *XDR: External Data Representation Standard*. Network Working Group, Aug. 1995. (published as RFC 1832).

[9] Sun Microsystems. *Jini Architecture Specification*, Oct. 2000. Version 1.1.

[10] The Open Group. *DCE 1.1: Remote Procedure Call*. Berkshire, UK, Aug. 1997.

[11] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic timing model generation by CFG partitioning and model checking. In *Proc. Design, Automation and Test in Europe (DATE'05)*, Munich, Germany, Mar. 2005.