SONY

OPEN-R SDK

Programmer's Guide

115-01

Notes on This Document

Notes on Using This Document

- The contents provided by this document (PDF files) are intended only for supplying information.
- The contents provided by this document (PDF files) are subject to change without notice.
- We are not responsible for errors or omissions in technical or editorial aspects concerning the contents described in this document. We also are not responsible for technical measures, correspondence, execution according to this document, as well as for the results occurred by them such as inevitable, indirect or incidental damages.

Notes on Copyright

- Sony Corporation is the copyright holder of this document.
- No information in this document may be duplicated, reproduced or modified. It is also prohibited to publish the contents of this document on the Internet Website or other public media without the express written permission of Sony Corporation.

About Trademarks

- AIBO and OPEN-R is a trademark or a registered trademark of Sony Corporation.
- "Memory Stick" is a trademark of Sony Corporation. "TM" is not described in this document.
- Microsoft and Windows are a registered trademark of Microsoft Corporation in the United States and/or other countries.
- UNIX is a registered trademark of The Open Group in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- MIPS is a registered trademark of MIPS Technologies, Inc. in the United States and/or other countries.
- Adobe Acrobat and Adobe Reader are registered trademarks of Adobe Systems Incorporated.
- Other system names, product names, service names and firm names contained in this document are generally trademarks or registered trademarks of respective makers.

Index

Chapter1 Basic Knowledge

1.1 OPEN-R

"OPEN-R" is the interface that Sony is promoting for the entertainment robot systems to expand the capabilities of entertainment robots. This interface is layered and optimized to enable efficient development of hardware and software for robots.

The OPEN-R SDK discloses the specifications of the interface between the ‘system layer’ and the ‘application layer’. These specifications are essential knowledge for developing robot software.

Features of OPEN-R software:

☐ Modularized software and inter-object communication

OPEN-R software is object-oriented and modular. Software modules are called "objects" (specifically, “OPEN-R objects”).

In OPEN-R, robot software is implemented so that processing is performed by multiple objects with various functionality running concurrently and communicating each other via inter-object communication.

Connections between objects are defined in an external description file. When the system software boots, the description file is loaded and used to allocate and configure the communication paths for inter-object communication. Connection ports in objects are identified by the service name, which enables objects to be highly modular and easily replaceable as software components.

☐ Layered structure of the software and services provided by the system layer

The OPEN-R system layer provides a set of services (input of sound data, output of sound data, input of image data, output of control data to joints, and input of data from various sensors) as the interface to the application layer. This interface is also implemented by inter-object communication.

These services enable application objects to utilize the robot’s underlying functionality, without requiring detailed knowledge of the hardware devices that comprise the robot.

The system layer also provides the interface to the TCP/IP protocol stack, which enables programmers to create networking applications utilizing the wireless LAN.

1.2 Object

OPEN-R application software consists of several OPEN-R objects. (Whenever you see the word “object” in this manual, it means an “OPEN-R object”, not an object in the traditional C++ sense of the word.) The concept of an object is similar to one of a process in the UNIX or Windows operating systems with regard to the following points of view.

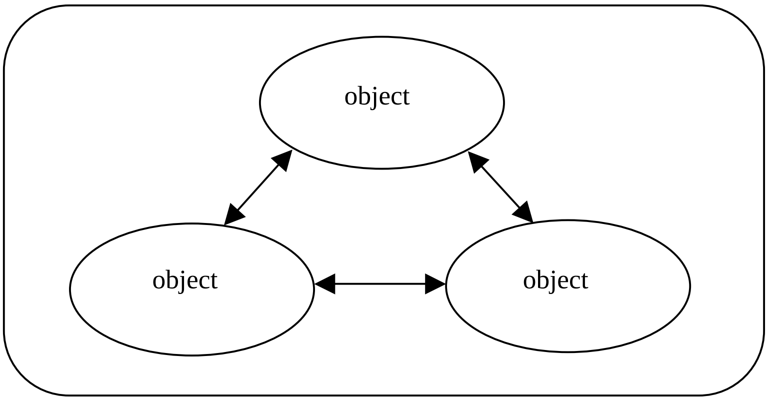


Fig1 OPEN-R Application software

- ☐ **An object corresponds to one executable file.**
  An object is a concept that only exists at run-time. Each object has a counterpart in the form of an executable file, created at compile-time. Source code is compiled and linked to create this executable file. Then, the file is put on an AIBO Programming Memory Stick. When AIBO boots, the system software loads the file from the AIBO Programming Memory Stick and executes it as an object. (An executable file usually has a filename with an extension of ".bin".)

- ☐ **Each object runs concurrently with other objects.**
  Each object has its own thread of execution and runs concurrently with other objects in the system.

The following are characteristics specific to objects.

- ☐ **Objects exchange information using message passing**
  An object can send messages to other objects. A message contains some data and a selector, which is an integer that specifies a task to be done by the receiver of the message. When an object receives a message, the function corresponding to the selector is invoked, with the data in the message as its argument. A function corresponding to a selector is called a “method”.

  An important feature of objects is that they are single-threaded. This means an object can process only one message at a time. If an object receives a message while it is processing another message, the second message is put into the message queue and processed later.

  Below is the typical life cycle of an object:

  1 Loaded by the system
  2 Wait for a message
  3 When a message arrives, execute the method corresponding to the selector specified in the message. Possibly send some messages to other objects.
  4 When the method finishes execution, go to step 2.

Note that this is an infinite-loop: an object cannot terminate itself. It persists while the system is activated.

- ☐ **An object has multiple entry points**

  Unlike an ordinary programming environment in which a program has a single entry point "main()", OPEN-R allows an object to have multiple entry points. Each entry point corresponds to a selector as explained above. Some entry points have purposes that are determined by the system, e.g. initialization and termination. Other entry points have purposes specific to the object.

  An object also has a special entry point called Prologue(). This entry point does not correspond to a selector. It is executed once when the object is loaded by the system. Prologue() does some initialization and calls constructors for global variables.

1.3 Inter-object communication

Software controlling entertainment robots typically consists of various objects, each with its own tasks, such as image recognition, speech recognition, motion control and motion generation. They communicate with each other while they perform their tasks. In OPEN-R, this communication between objects is called "inter-object communication.".

Communication between subjects and observers

The use of inter-object communication enables each object to be created separately and later be connected to other objects. This results in a very efficient development lifecycle.
When two objects communicate, the side that sends data is called the "subject," and the side that receives data is called the "observer". The subject sends a 'NotifyEvent' to the observer. NotifyEvent includes the data that the subject wants to send to the observer. The observer sends a 'ReadyEvent' to the subject. The purpose of ReadyEvent is to inform the subject that the observer is ready to receive data or not. If the observer is not ready to receive data, the subject does not send any data to the observer.

Fig2 shows a case where the subject of object A communicates with the observer of object B.

Before the observer receives data from the subject, the observer must inform the subject of its current state. When the observer is in a state ready to receive data, the observer sends 'ASSERT-READY' to the subject. When the observer is in a state not ready to receive data, the observer sends 'DEASSERT-READY' to the subject. When the subject receives ASSERT-READY from the observer, the subject starts to send data to the observer. After the observer receives this data and is ready to receive the next data, the subject sends ASSERT-READY again.

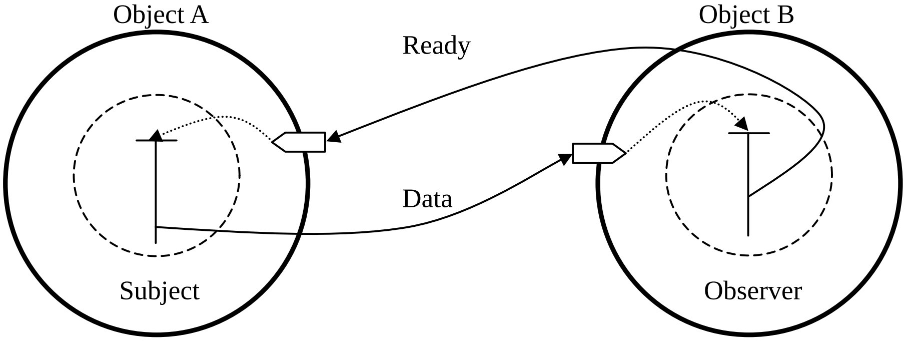


Fig2 Inter-object communication

Chapter2 OPEN-R Programming

2.1 Flow of development

The following illustrates the flow of development using the OPEN-R SDK.

(1) Design your objects

You must design the functions of the objects you are going to create, as well as the flow of data between these objects. For example, in the case of designing an object for tracking a pink ball, the object must receive image data, it must detect a pink ball in the image, and it must send command data for moving the head toward the ball.

(2) Design the data type for inter-object communication

You must choose an appropriate data type to carry messages between objects. For example, you can choose OFbkImageVectorData for input data and OCommandVectorData for output data.

(3) Description of stub.cfg

The connection between the entry points (where the message is received) and the actual member functions of the core class is described with specified form in a file called stub.cfg (stub configuration file).

In stub.cfg, we also describe the services that send and receive data to/from other objects. By executing a tool named ‘stubgen2’, the intermediate files are created from stub.cfg, which are needed for compiling purposes.

(4) Implementation of a core class

You must implement the core class of your object, which includes the functions specified by stub.cfg, the four DoXXX() member functions (which are always implemented on the core class) and other member functions.

(5) Decide the configuration of your .ocf file

The configuration of objects at run-time is specified here.

(6) Build

You build an executable file for your object by linking with other necessary libraries.

(7) Edit the setting files

You must edit the setting files needed at run-time. The following are the relevant setting files.

OBJECT.CFG	Description of objects to be executed
CONNECT.CFG	Description of connections between objects
DESIGNDB.CFG	Description of files with paths, which are accessed by objects at run-time

(8) Execution on AIBO

The following files are copied to the specified directory on an AIBO Programming Memory Stick.

- OPEN-R directory
- Execution file (.BIN)
- Edited setting files
- Data files such as motion and sound

After creating a wireless LAN environment for connecting AIBO with your PC, insert an AIBO Programming Memory Stick into AIBO and boot it.

(9) Debugging

If you happen to have bugs in your program, you can debug them by using special debugging macros, such as OSYSPRINT and OSYSLOG1, as well as other error information, from the messages that are printed in the wireless LAN console.

2.2 Core class

A core class is a C++ class that represents an object. Each object should be represented by only one core class.

Objects have entry points for receiving messages. As shown in figure 1-3, each entry point corresponds to a particular method of the object, and each method corresponds to a particular member function of the core class. Core classes have the following characteristics.

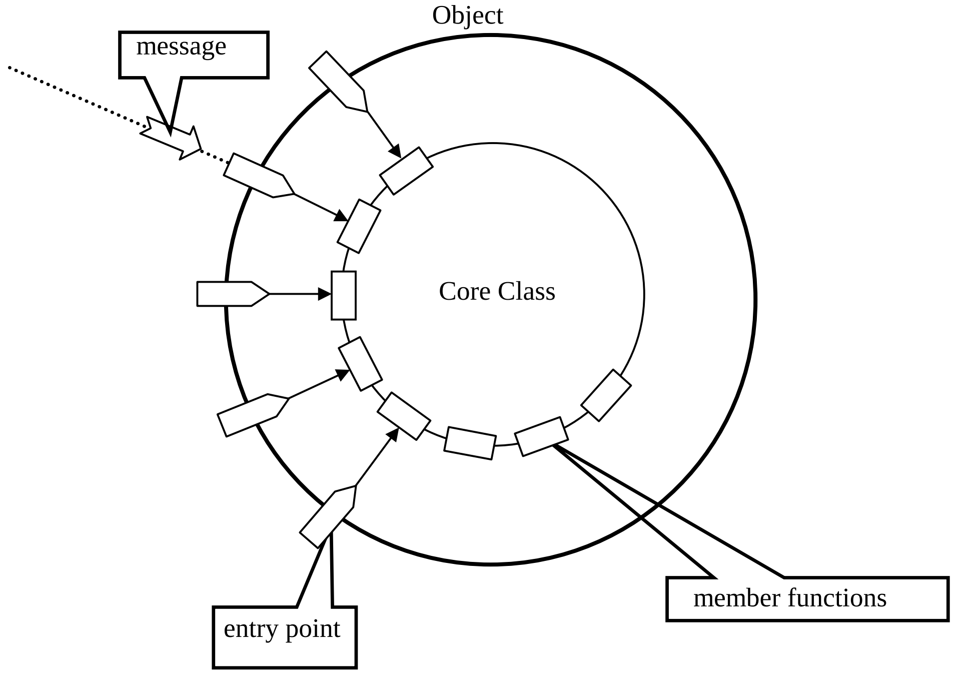


Fig3 Core class

Characteristics of core classes

- ❑ A core class inherits from the OObject class.
- ❑ A core class implements DoInit(), DoStart(), DoStop(), DoDestroy().
- ❑ A core class has the necessary number of OSubject and OObserver.

  Example
  OSubject* subject[numOfSubject];
  OObserver* observer[numOfObserver];

  numOfSubject and numOfObserver are defined in def.h that is generated by the stubgen2 command.

- ❑ Some member functions in the core class correspond to specific methods in the object:

(1) Methods that are called at startup and shutdown:

Init method
This is called at startup. This method initializes instances and variables.

Start method
This is called at startup after Init is executed in all objects.

Stop method
This is called at shutdown.

Destroy method
This is called at shutdown after Stop is executed in all objects. This method destroys the subject and observer instances.

The Init method, Start method, Stop method, and Destroy method correspond to each DoInit(), DoStart(), DoStop() and DoDestroy() function in the object's corresponding core class, respectively.

(2) When a message is received from another object, the following methods are used.

- Methods used in subjects:

Control method	This receives the connection results between the subject and its observers.
Ready method	The subject receives ASSERT-READY or DEASSERT-READY notifications from the observers.

- Methods used in observers:

Connect method	This receives the connection results between an observer and its subjects.
Notify method	This receives a message from the subject.

These methods have the following characteristics.

(a) The member functions corresponding to Control methods, Ready methods, Connect methods, and Notify methods are described in stub.cfg.

(b) The member functions receiving a message are described in stub.cfg, but it is not necessary to describe the member functions sending a message in stub.cfg.

Below, we will show some examples to illustrate these concepts. The first example, SampleClass1, inherits from OObject. The four standard OPEN-R member functions (DoInit(), DoStart(), DoStop() and DoDestroy()) are defined here. The following is the definition of SampleClass1.

```
#include <OPENR/OObject.h>

class SampleClass1 : public OObject {
public:
      SampleClass1();
      virtual ~SampleClass1() {}

      virtual OStatus DoInit(const OSystemEvent& event);
      virtual OStatus DoStart(const OSystemEvent& event);
      virtual OStatus DoStop(const OSystemEvent& event);
      virtual OStatus DoDestroy(const OSystemEvent& event);
};
```

The second sample, SampleClass2, communicates with other objects. We must define the necessary number of OSubjects and OObservers. The necessary number is described in def.h. In the following sample, OSubject and OObserver are used to perform inter-object communication.

```
#include <OPENR/OObject.h>
#include <OPENR/OSubject.h>
#include <OPENR/OObserver.h>
#include "def.h"
class SampleClass2 : public OObject {
public:
    SampleClass2();
    virtual ~SampleClass2() {}

    OSubject*  subject[numOfSubject];
    OObserver* observer[numOfObserver];

    virtual OStatus DoInit(const OSystemEvent& event);
    virtual OStatus DoStart(const OSystemEvent& event);
    virtual OStatus DoStop(const OSystemEvent& event);
    virtual OStatus DoDestroy(const OSystemEvent& event);

      //Describe the member functions corresponding to Notify,
      //Control, Ready, Connect method.

};
```

2.3 Stub

A stub is used to connect an entry point of an object with a member function in a core class. The stub is defined in xxxStub.cc, which is automatically generated from stub.cfg, by a stub generator (the stubgen2 command). Only one instance of a core class is generated as a global variable in xxxStub.cc.

The following items are described in stub.cfg.

- The number of subjects and the number of observers
- Services used in inter-object communication

The subjects and observers provide the services for inter-object communication. Each service has a unique name, in order to distinguish that service from other services in the system. You can connect the subject's service to the observer's service by describing both service names in CONNECT.CFG.

The following is a sample stub.cfg file.

```
ObjectName : SampleClass
NumOfOSubject   : 1
NumOfOObserver  : 2
Service : "SampleClass.Func1.Data1.S", Control(), Ready()
Service : "SampleClass.Func2.Data2.O", Connect(), Notify1()
Service : "SampleClass.Func3.Data2.O", null, Notify2()

Extra : UpdatePowerStatus()
```

Here are the descriptions of each item:

❑ **ObjectName**
A core class name.

❑ **NumOfOSubject**
This is the number of subjects. You must specify at least 1 subject. When you do not need a subject in your program, you should register one dummy subject.

❑ **NumOfOObserver**
This is the number of observers. You must specify at least 1. When you do not need an observer in your program, you should register one dummy observer.

❑ **Service**
Here, you specify the communication service for the object. A service corresponding to each subject and observer is described. A service consists of the following items below.

"(Connection name)", (Member function 1), (Member function 2)

- **Connection name**

The connection name consists of the following items.
(Object name).(Subname).(Data name).(Service type)

Object Name
You can use any name you like, but this is usually the core class name.

Subname
This is a service name and must be unique. Do not use the same subname that other services use.

Data name
This is the name corresponding to the data type used in inter-object communication.

Service type
S(subject) or O(observer) is specified.

- **Member function 1**
  This member function is called when a connection result is received. You can freely use any names for this function. This function is implemented in the core class. In case you do not need it, you can specify "null" here.

- **Member function 2**
  If this service is for observers, this function is called when a message is received from a subject. If this service is for subjects, this function is called when ASSERT-READY or DEASSERT-READY is received from an observer. You can use any name you like for this function. This function is implemented in the core class. In case you do not need it, you can specify "null" here.

The following is an example of a "dummy subject" service.

```
Service : "SampleClass.DummySubject.DontConnect.S", null, null
```

You can reuse the same member functions in other services.
The following is an example.

```
Service : "SampleClass.Out1.Data1.S", Control(), Ready()
Service : "SampleClass.Out2.Data1.S", Control(), Ready()
```

Control() and Ready() are the member functions common to the two services.

- ☐ **Extra entry**
  In case additional entry points (other than Init, Start, Stop, Destroy, Control, Ready, Connect, and Notify) are needed, you can add them by using an 'Extra' entry. For example, a call-back function used in network communication is one example:

```
     Extra : NewExtra1()
     Extra : NewExtra2()
```

Next, we explain how messages are sent and received between the subject "a" of object A and the observer "b" of object B, using Fig1-4 and the sample description of stub.cfg.

The following is the stub.cfg of object A.

```
ObjectName : ObjectA
NumOfOSubject   : 1
NumOfOObserver  : 1
Service : "ObjectA.SendString.char.S", null, subject_a()
Service : "ObjectA.DummyObserver.DontConnect.O", null, null
```

The following is the stub.cfg of object B.

```
ObjectName : ObjectB
NumOfOSubject   : 1
NumOfOObserver  : 1
Service : "ObjectB.DummySubject.DontConnect.S", null, null
Service : "ObjectB.ReceiveString.char.O", null, observer_b()
```

The connection of service between subject a and observer b is described in CONNECT.CFG as follows.

```
ObjectA.SendString.char.S     ObjectB.ReceiveString.char.O
```

The following are the steps of sending and receiving a message.

(1) DoStart() in object B sends ASSERT-READY to the subject in object A. This notification reaches subject_a() of the core class in object A.

(2) subject_a() sends a message to Object B. This notification reaches observer_b() of the core class in object B.

(3) When observer_b() requests Object A to send the next message, observer_b() sends ASSERT-READY to Object A.

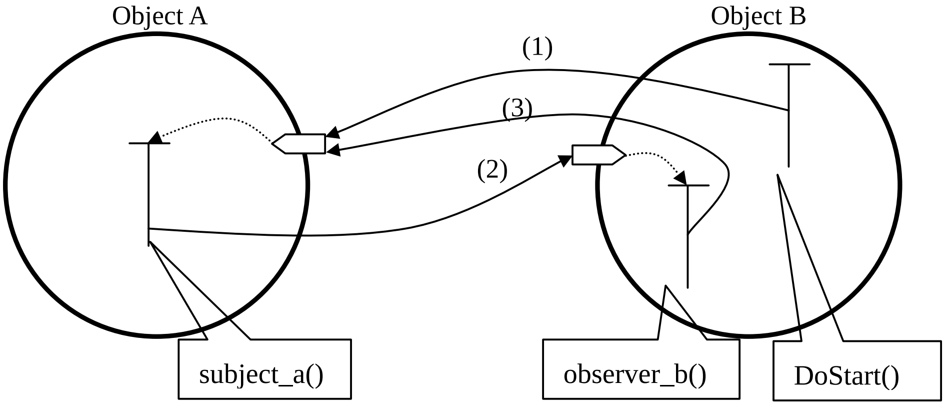


Fig4 Example of an inter-object communication

2.4 DoInit(),DoStart(),DoStop(),DoDestroy()

Core classes should override DoInit(), DoStart(), DoStop(), and DoDestroy() and describe the processes which are unique to each core class.

- **DoInit**()

  DoInit() is called on startup.
  DoInit() registers the communication services.

  The following are macro that simplify this description.

  NEW_ALL_SUBJECT_AND_OBSERVER
  This registers the necessary number of observers and subjects.

  REGISTER_ALL_ENTRY
  This registers the connection to services offered by other objects.

  SET_ALL_READY_AND_NOTIFY_ENTRY
  This registers the entry points for receiving messages.

  The following is a sample:

```
OStatus
 SampleClass::DoInit(const OSystemEvent& event)
    {
       NEW_ALL_SUBJECT_AND_OBSERVER;
       REGISTER_ALL_ENTRY;
       SET_ALL_READY_AND_NOTIFY_ENTRY;
    }
```

- **DoStart**()

  After DoInit() is executed in all the objects, DoStart() is called. Here, each observer usually sends ASSERT-READY to its connecting subjects.

  In case you want to delay sending ASSERT-READY to specific subject, you can send it in another member function, instead of DoStart(). The following are macros that simplify this description.

  ENABLE_ALL_SUBJECT
  This enables the subjects of the core class.

  ASSERT_READY_TO_ALL_OBSERVER
  This sends ASSERT_READY to all the connecting subjects.

  The following is an example using these macros.

```
OStatus
SampleClass::DoStart(const OSystemEvent& event)
{
   ENABLE_ALL_SUBJECT;
   ASSERT_READY_TO_ALL_OBSERVER;
   return oSUCCESS;
}
```

- **DoStop()**
  This is called at shutdown. Each observer usually sends DEASSERT-READY to its connecting subjects. The following are macros that simplify the description.

  DISABLE_ALL_SUBJECT
  This disables the subjects of the core class.

  DEASSERT_READY_TO_ALL_OBSERVER
  This sends DEASSERT_READY to all the connecting subjects.

  The following is a sample code using the macros.

```
OStatus
SampleClass::DoStop(const OSystemEvent& event)
{
   DISABLE_ALL_SUBJECT;
   DEASSERT_READY_TO_ALL_OBSERVER;
   return oSUCCESS;
}
```

- **DoDestroy**()
  This is called at shutdown, after DoStop() is called on all objects. The following is a macro that simplifies this description.

  DELETE_ALL_SUBJECT_AND_OBSERVER
  This deletes all observers and subjects which are generated in DoInit().

  The following is a sample code using these macros.

```
OStatus
SampleClass::DoDestroy(const OSystemEvent& event)
{
  DELETE_ALL_SUBJECT_AND_OBSERVER;
  return oSUCCESS;
}
```

2.5 Sending and receiving data

When an observer receives data, the observer sends an ASSERT-READY message to the subject.

```
void
SampleObserver::SendAssertReady()
{
   observer[obsFunc1]->AssertReady();
}
```

When a subject receives an ASSERT-READY message from an observer, the subject then sends data to the observer.

```
void
SampleSubject::Ready(const OReadyEvent& event)
{
   char str[32];
   strcpy(str, “!!! Hello world !!!”);
   subject[sbjFunc2]->SetData(str, sizeof(str));
   subject[sbjFunc2]->NotifyObservers();
};
```

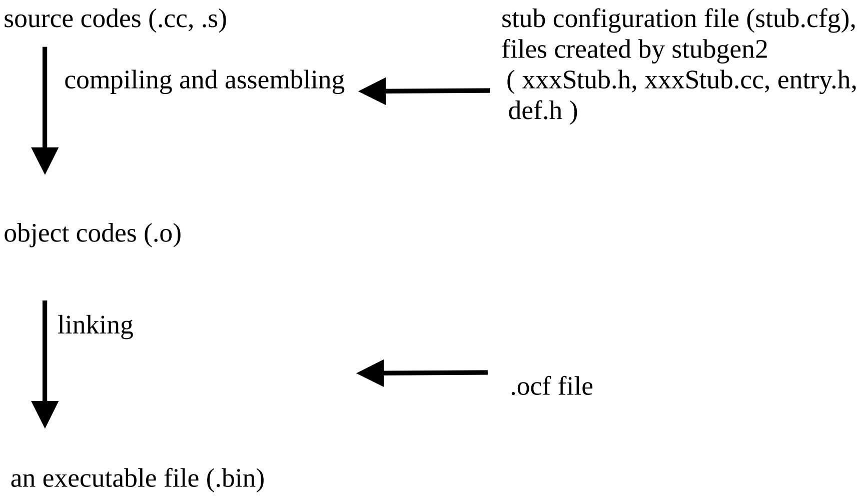
When an observer receives data from a subject and then is in a state to receive data again, the observer sends ASSERT-READY to the subject.

```
void
SampleObserver::Notify(const ONotifyEvent& event)
{
  const char* text = (const char*)event.Data(0);
  observer[obsFunc1]->AssertReady();
}
```

obsFunc1 and sbjFunc2 are index numbers for the arrays observer[] and subject[], espectively, to identify the observer and the subject. They are defined in def.h, which is automatically generated from stub.cfg by executing the "stubgen2" command.

Chapter3 Building

Below is the flow of building objects and creating an executable file.


source codes (.cc, .s)
compiling and assembling
stub configuration file (stub.cfg), files created by stubgen2 (xxxStub.h, xxxStub.cc, entry.h, def.h)
object codes (.o)
linking
.ocf file
an executable file (.bin)

3.1 .ocf

A file with an ".ocf" extension is used to specify the configuration of the object. An .ocf file must have the following format.

object OBJECT_NAME STACK_SIZE HEAP_SIZE SCHED_PRIORITY CACHE TLB MODE

Example:

```
object helloWorld 3072 16384 128 cache tlb user
```

You can specify the following parameters:

- **OBJECT_NAME**
  The name of the object.

- **STACK_SIZE**
  The size of the stack, in bytes, for the object. The stack is not extended at runtime. If an object uses more stack space than the specified size (by declaring large amounts of automatic variables, doing deeply nested function calls, etc.), the result is undefined.

- **HEAP_SIZE**
  If an object runs out of heap space, the heap is extended by this amount (in bytes). A heap is an area for allocating memory requested by the malloc() or new operator.

  Because extending the heap is a costly operation, specifying a HEAP_SIZE value that is too small slows down the object by forcing frequent heap resizing.

- **SCHED_PRIORITY**
  The scheduling priority of the object, represented as an eight-bit unsigned integer. The upper four bits specify the scheduling class; an object with a lower class never runs while an object with a higher class is running. The

recommended scheduling class for an ordinary application is 8 (128 in scheduling priority) or lower. The lower four bits control the ratio of execution time between objects within the same scheduling class. The higher the value, the more time the object gets.

- **CACHE**
  Specify “cache” or “nocache”. If “nocache” is specified, cache memory in the processor is disabled while executing the object. Specifying “cache” is normally recommended.

- **TLB**
  Specify “tlb” or “notlb”. If “tlb” is specified, the memory area for the object is allocated in the virtual address space. Otherwise, the memory area is allocated in the physical address space (KSEG0 or 1). This value is ignored when a “nomemprot” configuration is used. Specifying “tlb” is recommended. You must specify “tlb” if you specify “user” for MODE.

- **MODE**
  Specify “kernel” or “user”. If “user” is specified, the processor's execution mode is set to user mode when the object is executed. Otherwise, the object is executed in kernel mode. This value is ignored when a “nomemprot” configuration is used. An object always runs in kernel mode in “nomemprot” configuration.

3.2 StubGenerator

The “Stubgen2” command reads stub.cfg and generates intermediate files to connect the methods of an object with the member functions of a core class.
def.h is one file that is generated by Stubgen2. The following is the sample code of def.h.

```
const int sbjFunc1 = 0;
const int obsFunc2 = 0;
```

“sbj” or “obs” is added in the head of a subname described in stub.cfg.
The following is the definition of SampleClass.h.

```
OSubject*     subject[numOfSubject];
OObserver*   observer[numOfObserver];
```

---

sbjFunc1 and obsFunc2 are the indexes of the arrays to specify a subject and an observer (Examples: subject[sbjFunc1]; observer[obsFunc2]).

3.3 Compiling and assembling

Use the following command for compiling and assembling. The command is common to both the C++ source files(.cc) and the assembly language source files (.s).

```
/usr/local/OPEN_R_SDK/bin/mipsel-linux-gcc -c [other options] FILE
```

This creates an object file named XXX.o, where XXX is the base name of FILE. Be sure to specify the -c option; without -c, gcc attempts to link files to form an executable, and it fails. gcc for the OPEN-R SDK cannot perform linking. You should use the mkbin command instead.

You can specify directories for header files using the -I option if necessary. For example, you can include the OPEN-R header files using the following options.

```
-I/usr/local/OPEN_R_SDK/OPEN_R/include/R4000
-I/usr/local/OPEN_R_SDK/OPEN_R/include
```

3.4 Linking

Use the following command for linking object files.

```
/usr/local/OPEN_R_SDK/OPEN_R/bin/mkbin -o XXX.bin XXX.ocf
[other options] AAA.o BBB.o CCC.o ...
```

This links the object files AAA.o, BBB.o, CCC.o ... and creates an executable file XXX.bin. A file with `.ocf' extension is used to specify the configuration of the object.

Other options for mkbin, except listed below, are passed to the linker that is called by mkbin. Refer to the manual of GNU ld for the options that can be passed to the linker. In addtion to the options which are specified by the command line, mkbin also passes the options for linking the default libraries to the linker. You can see what option is passed to the linker by running mkbin with the -v flag.

mkbin creates several intermediate files that are not removed automatically. Their names are listed in the description of the –o flags below. Do not use the same name for your own file or it will be overwritten.

Below is a list of options processed by mkbin itself.

-o PATH
Specifies the path of the generated executable file. The default value is a.bin.

PATH is also used to determine the names of intermediate files, as follows:

1. Remove trailing .bin, if it exists.
2. Append one of the following strings to the result of 1.

.snap.cc, .snap.o, .nosnap.elf, .snap.elf, .rel.elf

-m FILE
This specifies the FILE to be treated as an .ocf file regardless of its suffix.

-p PATH
This specifies the directory where the SDK is installed. The default value is /usr/local/OPEN_R_SDK.

--nodefaultlib
This inhibits linking of the default libraries.

--nocrt

This inhibits linking of the startup routines. In case this option is not specified, the following file is automatically linked.

crtbegin.o
crtend.o

--novm

This specifies that the generated executable file is used with `nomemprot' configuration.  The effect is the same as specifying `notlb' for TLB and `kernel' for MODE in .ocf file.

-v

This prints verbose messages.

Chapter4 Running

4.1 Selection of OPEN-R configuration

There are several OPEN-R configurations you can choose from to run your program in. The OPEN-R directory corresponding to each configuration is installed in the directory /usr/local/OPEN_R_SDK/OPEN_R/MS/. When you run your object on a robot, copy the OPEN-R directory corresponding to the configuration that you chose, as well as the objects that you created, to an AIBO Programming Memory Stick.

BASIC/memprot/OPEN-R/
BASIC/nomemprot/OPEN-R/
WLAN/memprot/OPEN-R/
WLAN/nomemprot/OPEN-R/
WCONSOLE/memprot/OPEN-R/
WCONSOLE/nomemprot/OPEN-R/

4.1.1 Differences among BASIC/WLAN/WCONSOLE

Choose one of the following depending on your purpose.

❑ **BASIC**
Without a wireless LAN environment

❑ **WLAN**
With a wireless LAN environment and without a wireless console

❑ **WCONSOLE**
With both a wireless LAN environment and a wireless console

4.1.2 memprot/nomemprot

Choose one of the following depending on your purpose.

❑ **memprot**
This configuration enables memory protection. Objects can work with user/tlb mode. Refer to “3.1 .ocf” for user/tlb mode details.
The memory area that objects in user/tlb mode uses is placed in a virtual address space using TLB(translation lookaside buffer). This has the following advantages and disadvantages.

Advantages

- An object running in user mode cannot access the memory area of other objects. Thus, it is easier to find bugs caused by using a wrong pointer value.

- In a virtual address space, you can use physically fragmented memory areas as a continuous memory area. This increases memory utilization.

Disadvantages

- Shared memory is reserved by 4096 byte chunks. When you reserve large quantities of a small memory area, memory utilization is decreased.

- When the following happens, some overhead is incurred which affects the speed of your application:

  (a) The conversion from a virtual address to physical address in case of a TLB miss.
  (b) The allocation of shared memory, the release of shared memory, and the attachment of shared memory to other objects

- **nomemprot**
  This deactivates memory protection. Here, all objects work with kernel/notlb mode. This has the opposite advantages and disadvantages compared to memprot.

  In the nomemprot environment, an object can access the memory area of other objects without using shared memory. However, if you use the API for shared memory in a nomemprot environment, your code can be used both in memprot and nomemprot environments without any changes. In the nomemprot environment, most of the API functions for managing shared memory are implemented as dummy functions, which work at high speed.

4.2 Files to be set

Some setting files should be customized to run your object(s) on AIBO. The following shows how to write files such as OBJECT.CFG, CONNECT.CFG and DESIGNDB.CFG.
These files are located in the directory OPEN-R/MW/CONF on the AIBO Programming Memory Stick.

4.2.1 OBJECT.CFG

Enumerate the executable files corresponding to the object(s) that you want to execute.

```
/MS/OPEN-R/MW/OBJS/XXX.BIN
/MS/OPEN-R/MW/OBJS/YYY.BIN
```

/MS/ indicates the root directory on an AIBO Programming Memory Stick.

4.2.2 CONNECT.CFG

The connection between a subject and an object is described.

```
Class1.Func1.Data1.S Class2.Func2.Data1.O
Class1.Func3.Data2.S Class3.Func4.Data2.O
```

Each line includes the following items.
"a subject service" (.S) ,"a space", "an observer service (.O)"

The data name in the subject service and the data name in the observer service must be the same.

4.2.3 DESIGNDB.CFG

You can change the files to be loaded depending on the type of a robot that your program is running on. We describe keywords to find the file.

```
#
# DESIGNDB.CFG
#

[ERS-210]
KEYWORD1   /MS/OPEN-R/MW/CONF/ERS-210.TXT
KEYWORD2   /MS/OPEN-R/MW/DATA/P/210.WAV

[ERS-220]
KEYWORD1   /MS/OPEN-R/MW/CONF/ERS-220.TXT
KEYWORD2   /MS/OPEN-R/MW/DATA/P/220.WAV
```

/MS/ indicates the root directory in an AIBO Programming Memory Stick.

A keyword is used as a tag to identify each file, and it can be any word you like. This keyword is referred to when a data file is read using the OPEN-R API, as shown below.

```
OPENR::FindDesignData (“KEYWORD1”,
      (ODesignDataID*)&memID, &addr, &size)
```

4.3 Execution on AIBO

4.3.1 Creation of AIBO Programming Memory Stick

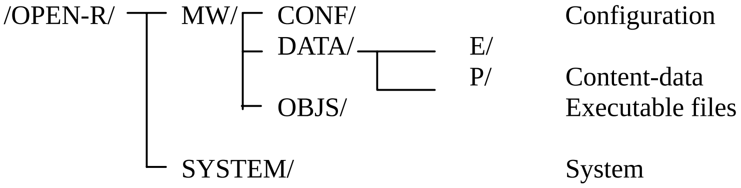
First, you should copy the OPEN-R directory to the root of an AIBO Programming Memory Stick. Then, you copy executable files (xxx.bin) to the AIBO Programming Memory Stick. The executable files are copied to the following directory.

/OPEN-R/MW/OBJS/

Content data such as LED, motion, and sound data are copied to the following directory on the AIBO Programming Memory Stick.

/OPEN-R/MW/DATA/P/

OPEN-R directory



4.3.2 Execution of AIBO Programming Memory Stick

After copying the OPEN-R directory and editing necessary files to your AIBO Programming Memory Stick, insert it into AIBO and press the pause button.

Notes

AIBO may not boot if the remaining battery capacity is low.

Chapter5 Debug

5.1 Error log output

We usually use printf and cout to output characters for debugging. In the OPEN-R environment, several objects run concurrently, and there is a chance that the displayed characters from each object are mixed together on-screen. We recommend that you use the macros OSYSPRINT and OSYSLOG to avoid this problem.

In OSYSPRINT() and OSYSDEBUG(), you can specify the same arguments as ones used in printf(). When the symbol "OPENR_DEBUG" is not defined, OSYSDEBUG() is replaced with a null string. OPENR_DEBUG is defined by using the compiler option–DOPENR_DEBUG.

```
OSYSPRINT((“test: %d\n”, x, y));
OSYSDEBUG((“Hello \n”));
```

The max length of the displayed characters in OSYSPRINT and OSYSDEBUG is 243 characters including the terminating null character.

OSYSLOG1 is a macro for displaying errors. An error level is specified as the first argument. In OSYSLOG1, a line feed is automatically added at the end of the string.

```
OSYSLOG1((osyslogERROR, “This is error!”));
```

The following characters are displayed as the result of OSYSLOG1.

```
[oid:80000043,prio:1] This is error!
```

The numeric value after "prio:" is the error level specified in the first argument. You can specify the following error levels for the first argument of OSYSLOG1.

prio	oid/prio	Value of prio
osyslogERROR	Displayed	1
osyslogWARNING	Displayed	2
osyslogINFO	Displayed	3

5.2 Investigating cause of a CPU exception

When a program performs a wrong operation such as referring to an invalid address or executing an invalid instruction, a CPU exception may occur and execution of the program is terminated. If this happens, a special sound is emitted and information necessary for investigating the cause of the CPU exception is written to the AIBO Programming Memory Stick. This section describes how to investigate the cause of the CPU exception using this information.

The CPU used in AIBO is a MIPS processor. Understanding the MIPS architecture is required for investigating the cause of CPU exceptions.

5.2.1 Basic knowledge

❑ Operation mode used by OPEN-R
A CPU works in 32-bit mode and is ‘little endian’.

❑ The CPU exceptions that cause system shutdowns:
TLB exception
Address error exception
Reserved instruction exception
Floating-point exception
Coprocessor unusable exception
Bus-error exception

❑ How registers are used by the compiler
There are 32 generic registers in MIPS. The name in parentheses is a conventional name.

r0 (zero)
The value is always 0.

r2, r3 (v0, v1)
These are used to keep the value returned by a subroutine. v1 is used in case a 64-bit value is returned.

r4-r7 (a0-a3)
These are used to keep arguments for a subroutine. In case the number of arguments is more than 4, they are placed on the stack. When a member function in C++ is called, a0 stores the ‘this’ pointer.

r8-r15, r24, r25 (t0-t7, t8, t9)
These are used by subroutines without guaranteeing their persistence. Values stored in them may be destroyed by a subroutine call. For example, if you set t0 to 1 and then call a subroutine, it is not guaranteed that the value of t0 is 1 when returning from that subroutine.

r16-r23, r30 (s0-s7, s8)
These are temporarily used in a subroutine, but their original values are saved and restored before returning. Values stored in them after calling a subroutine are same as the values before the subroutine is called. For example, if you set s0 to 1 and call a subroutine, the value of s0 is 1 after returning from the subroutine.

r28(gp)
At the beginning of a subroutine, gp is set to the runtime address of symbol _gp. It is used to determine the runtime addresses of other symbols.

r29 (sp)
This is a stack pointer. A subroutine using a stack decrements the value of sp at the subroutine entry, and increases it to point to the previous position at its exit.

```
    3c8:      27bdffc8       addiu   sp,sp,-56
    ...
    440:      03e00008       jr      ra
    444:      27bd0038       addiu   sp,sp,56
```

r31 (ra)
This is used to store the return address by a subroutine. In case a subroutine is called with a jump and link instruction (jal, jalr), the return address is in the second place from the instruction. And a subroutine typically ends with "jr ra". Subroutines that call other subroutines usually save ra on the stack before calls.

The following are important registers in the system control coprocessor.

Status
This has the status such as the operating mode, interrupt mask, and so on.

Cause
This has the cause information of the last CPU exception.

BadVAddr
This has the virtual address of the cause when the CPU exception (TLB exception or address error exception) occurred.

EPC
This has the virtual address of an instruction that caused a CPU exception.

❑ How the stack is used by the compiler

The subroutine that uses "register s0-s8" or calls a subroutine uses the stack to save registers. A stack is sometimes used as a memory area for automatic variables. To call a subroutine, gcc and g++, the compilers used in the OPEN-R SDK, save the original "ra" to the highest address of the stack frame assigned for the call.

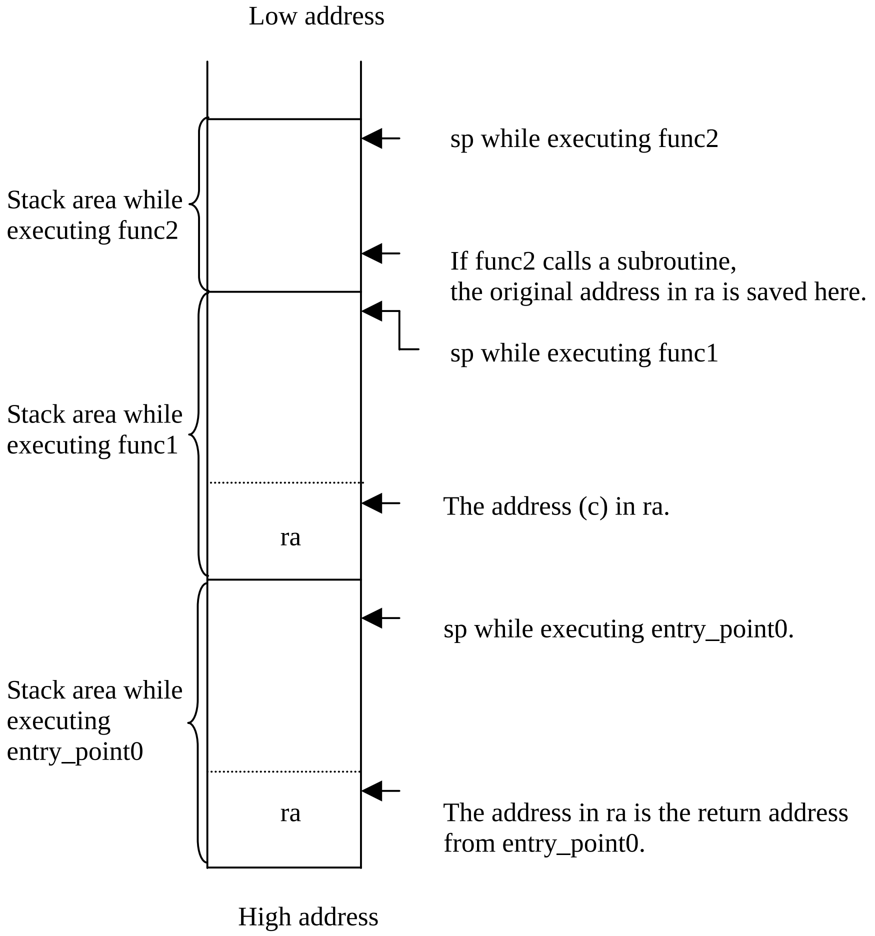
Example

```
void func2()
{
    char str[10];
    ...       <- address(a)
}

void func1()
{
    ...
    func2();
    ...       <- address (b)
}

int entry_point0()
{
    ...
    func1();
    ...       <- address (c)
}
```

The following is the state of the stack when executing the address (a) in func2.


Low address
sp while executing func2
Stack area while executing func2
If func2 calls a subroutine,
the original address in ra is saved here.
sp while executing func1
Stack area while executing func1
The address (c) in ra.
ra
sp while executing entry_point0.
Stack area while executing entry_point0
ra
The address in ra is the return address from entry_point0.
High address

Reference list
Please refer to the following books on the MIPS architecture.

Title : See MIPS Run
Writer : Dominic Sweetman
Publisher : Morgan Kaufmann Publishers
ISBN : 1558604103

Title : MIPS RISC Architecture
Writer : Gerry Kane, Joe Heinrich
Publisher : Prentice Hall
ISBN : 0135904722

5.2.2 System operation at CPU exception

In case a CPU exception causes a system shutdown, the system starts the Exception Monitor with the operation of all objects stopped. The wireless console is also stopped. The Exception Monitor executes the commands described in the file /OPEN-R/SYSTEM/CONF/EMON.CFG on the AIBO Programming Memory Stick, and turns off AIBO.

5.2.2.1 Default EMON.CFG

The file EMON.CFG is put in the following directory.
/OPEN-R/SYSTEM/CONF/

The content of the default EMON.CFG:

```
play exception
play warning
exception > /MS/OPEN-R/EMON.LOG
cp0 >> /MS/OPEN-R/EMON.LOG
cp1 >> /MS/OPEN-R/EMON.LOG
objs >> /MS/OPEN-R/EMON.LOG
dstack -max 0x4000 -r 0x40 >> /MS/OPEN-R/EMON.LOG
play finish
```

“/MS/” indicates the root directory of a Memory Stick. The output of the five commands, exception, cp0,cp1,objs, dstack are written to the /OPEN-R/EMON.LOG on the AIBO Programming Memory Stick. Sounds are output by the play commands during the Exception Monitor’s execution. The Exception Monitor is writing data during the output of a sound.

5.2.3 Identification of the type of CPU exception and object(s) affected

To identify the type of a CPU exception, look in EMON.LOG and find the exception code at the end of the exception command output.

To find an object, use the value of ‘[object info] context’ and the information of ‘[object list]’. For example, if you use the following EMON.CFG, the object "crash" is the cause of a CPU exception.

```
mCOOP exception
[exception info]
   time stamp: 0x000000004daf5139
   status:$12: 0x0000ff13, cause:$13: 0x00000010
 badvaddr: $8: 0x73692067,   epc:$14: 0x73692067
[object info]
 context: 0x80215240, state: 0x00000001
 last context: 0x00000000,last thread: 0x00000000
[initial value info]
 ..
 ..
[object list]
Name                   ContextID  ThreadID   ExecSpcID  OID        MetaSpace
-------------------------------------------------------------------------------
systemCore             0x80202c20 0x80202dc0 0x80202d80 ---------- mCore
systemCore(1)          0x80203a20 0x80203b80 0x80202d80 ---------- mCore
 ..
 ..
ovirtualRobot          0x80210240 0x8020d7a0 0x80208880 0x80000036 mCOOP
odesignedRobot         0x80211620 0x8020d260 0x80211040 0x80000037 mCOOP
osystemLogger          0x802114c0 0x8020d200 0x80211000 0x80000038 mCOOP
ovirtualRobotComm      0x80211360 0x80214980 0x80210fc0 0x80000039 mCOOP
ovirtualRobotAudioComm 0x80211200 0x80214920 0x80210f80 0x8000003a mCOOP
emergencyMonitor       0x80215660 0x80214860 0x80210f40 0x8000003b mCOOP
crash                  0x80215240 0x80214740 0x80210f00 0x8000003c mCOOP
[stack info]
```

5.2.4 Identification of an assembly instruction

When an address error exception or a TLB exception occurs, look at the value of BadVAddr first.

In case of an address error exception
→ BadVAddr has the address which failed to have a valid reference, such as an address which is not aligned.

In case of a TLB exception
→ BadVAddr has the virtual address which failed to have a valid translation.

Next, we look at the value of EPC. If the value of EPC is the same as the value of BadVAddr, we consider that it jumped to an invalid address and it caused the CPU exception. If the value of ra is a valid address, we look at the assembly instruction of the address with the following steps. If the value of EPC is different from the value of BadVAddr, we look at the assembly instruction of the address for the value of EPC. There are some possibilities that it is a data area or a stack area. If so, the program jumped to the data area or the stack area by mistake and it caused a CPU exception.
Using the value of ra where we can know the adjacent address before jumping, we can look at the assembly instruction.

How to find an assembly instruction
The address displayed by the Exception Monitor is determined at run-time. To look at the address, we calculate the address at link-time corresponding to that address. The following is the calculation necessary.

<Address at link-time > =
<Address at run-time> - <Value of gp register> + <Address of symbol _gp>

We look at the value of "[general register] gp:r28:" in EMON.LOG to know the value of the gp register.

To look at the address (at link-time) of symbol _gp, the following command is executed in the directory where you created the .bin file that caused the exception:.

```
% mipsel-linux-readelf -s <Object name>.nosnap.elf | grep '_gp$'
   471: 00466490     0 OBJECT  GLOBAL DEFAULT  ABS _gp
```

The value in the second place is the address, 0x00466490 in the example.

Hint
The following directory has a sample program, which automatically executes the steps described above. Refer to the top of the script for details.
sample/Crash.util/emonLogParser

After the address at link-time is found using this calculation, execute the following command.

```
% mipsel-linux-objdump -d -C <Object name>.nosnap.elf
```

After the result of disassembling the object is displayed, confirm the function name and the assembly instruction around the address at link-time. For example, if the address which you want to know is 4003d8, the place is in the function access_null_data_pointer().

```
% mipsel-linux-objdump -d -C crash.nosnap.elf
...
00000000004003bc <access_null_data_pointer()>:
  4003bc:       3c1c0006        lui     gp,0x6
  4003c0:       279c60d4        addiu   gp,gp,24788
  4003c4:       0399e021        addu    gp,gp,t9
  4003c8:       27bdffe0        addiu   sp,sp,-32
  4003cc:       afbc0010        sw      gp,16(sp)
  4003d0:       afbf001c        sw      ra,28(sp)
  4003d4:       afbc0018        sw      gp,24(sp)
> 4003d8:       8c050000        lw      a1,0(zero)
```

After the function name is found, examine the source code. Then, compare the result of the disassembling with the source code and find the place in question.

Notes

- If you specify the option –g in the compilation of an object, both the result of disassembling and source codes can be intermingled by replacing the argument –d in mipsel-linux-objdump with –S.
- An option –g is specified in the makefiles of the sample programs.
- The operation of objdump with an option –S is very slow.
- If the optimization level of compiling is low, it is easier to compare the disassembler result and the actual source code used. The optimization option used in the makefiles of the sample programs is –02.

If the address which you want to know is not in the disassembler results, see if the address is in the text area by running the following command.

```
% mipsel-linux-nm -n -C <Object name>.nosnap.elf
```

Then, confirm the content of the data area:

```
% mipsel-linux-objdump -st -C <Object name>.nosnap.elf
```

By [stack info] and [stack dump] in EMON.LOG, confirm the content of the stack.

5.2.5 Commands in EMON.CFG

exception

This command outputs the information of the object that caused the CPU exception and the contents of general registers.

Syntax

exception

Example

```
mCOOP exception
[exception info]
   time stamp: 0x0000000069cece90
   status:$12: 0x0000ff03, cause:$13: 0x00000008
 badvaddr: $8: 0x00000000,   epc:$14: 0x8064f8d8
[object info]
 context: 0x801d4560, state: 0x00000001
 last context: 0x00000000,last thread: 0x00000000
[initial value info]
 sr:$12: 0x0000ff03
[register dump]
 at: r1: 0x80000000, v0: r2: 0x8064f8bc, v1: r3: 0x00000000, a0: r4: 0x801d4560
 a1: r5: 0x69cec8e4, a2: r6: 0x801d4560, a3: r7: 0x801c1de0, t0: r8: 0x00000002
 t1: r9: 0x80000000, t2:r10: 0x0000ff03, t3:r11: 0x00000006, t4:r12: 0x00007f02
 t5:r13: 0x00000002, t6:r14: 0x81f438a8, t7:r15: 0x801d4562, s0:r16: 0x00000000
 s1:r17: 0x806a2384, s2:r18: 0x802154c8, s3:r19: 0x80443fc0, s4:r20: 0x696e6974
 s5:r21: 0x800e7268, s6:r22: 0x80448000, s7:r23: 0x00000000, t8:r24: 0x81f80000
 t9:r25: 0x8064f8bc, gp:r28: 0x806b5990, sp:r29: 0x80460238, s8:r30: 0x00000000
 ra:r31: 0x8064fed4, hi:   : 0x00000000, lo:   : 0x00000028

exception code:  2 TLB exception (load or instruction fetch)
```

Description

```
[exception info]
    status:$12:  The content of Status register
     cause:$13:  The content of Cause register
  badvaddr: $8:  The content of BadVAddr register
       epc:$14:  The content of EPC register
[object info]
      context:  Context ID of the object causing a CPU
               exception.  This is used to identify objects in
               the system.

[register dump]
               This displays the CPU exception code obtained
               by the Cause register and its explanation
```

cp0

This displays the contents of registers in coprocessor0 (system control coprocessor).

Syntax
cp0

Example

```
[system control co-processor registers]
   Index 0x00000001   Random 0x00000021 EntryLo0 0x0000011b EntryLo1 0x0000015b
 Context 0x00000000 PageMask 0x00000000    Wired 0x00000002      ??? ----------
BadVAddr 0x00000000    Count 0x8cc5b829  EntryHi 0x00000000  Compare 0x69d83f47
  Status 0x2000ff00    Cause 0x10008c2c      EPC 0x8009dca4     PRId 0x00002831
  Config 0x104766f3   LLAddr 0x1ca5214f      ??? ----------      ??? ----------
XContext 0x00000000      ??? ----------      ??? ----------      ??? ----------
     ??? ----------      ??? ----------      ECC 0x00000001 CacheErr 0xa420a100
   TagLo 0x00000000    TagHi 0x00000000 ErrorEPC 0x49105004      ??? ----------
```

cp1

This displays the contents of registers in coprocessor1 (floating-point unit).

Syntax
cp1

Example

```
[floating-point general registers]
 $f00: 0xdc9298c3 $f01: 0x94e10efb $f02: 0xfe3c2e6b $f03: 0xffffffff
 $f04: 0x59c427e6 $f05: 0xc17c3d19 $f06: 0x00000000 $f07: 0xc18c9c38
 $f08: 0x00000000 $f09: 0x40634000 $f10: 0x00000000 $f11: 0x4079b000
 $f12: 0x606a63be $f13: 0x3f7a98ef $f14: 0x00000000 $f15: 0x417c9c38
 $f16: 0x00000000 $f17: 0x4079b000 $f18: 0x00000000 $f19: 0xc18c9c38
 $f20: 0xd022208f $f21: 0xb9dcc68c $f22: 0x3e7a82a9 $f23: 0x0e0067a8
 $f24: 0x6793e6b2 $f25: 0x70b29eae $f26: 0xceae80e2 $f27: 0x1366b123
 $f28: 0xa1c0a478 $f29: 0xacc2a1ea $f30: 0x30b492ca $f31: 0x71f49005
[floating-point control registers]
  $0:fp_eir: 0x00002830
 $31:fp_csr: 0x01000000 (bit[17-0]: ----------------RN)
```

objs

This displays the list of objects which existed at the CPU exception. You can look at the relations between the name and the context ID of objects by the output of this command.
Syntax
objs

Example

```
[object list]
Name                   ContextID  ThreadID   ExecSpcID  OID        MetaSpace
----------------------------------------------------------------------------
systemCore             0x801c1de0 0x801c1f80 0x801c1f40 ---------- mCore
systemCore(1)          0x801c2be0 0x801c2d40 0x801c1f40 ---------- mCore
mCOOPReflector         0x801c63c0 0x801c7bc0 0x801c73c0 0x8000000b mCore
uniMailer              0x801c6520 0x801c7c20 0x801c7400 0x8000000c mCore
...
crash                  0x801d4560 0x801d3960 0x801d0100 0x8000003b mCOOP
```

stack, dstack

‘stack’ command displays the stack information of the object that caused a CPU exception. In addition, ‘dstack’ command displays the content of the stack. In case ‘dstack’ command is executed without arguments, the ‘dstack’ command dumps areas starting from the address pointed to by "sp". Using the option "-max", you can specify the maximum area to be dumped. If "-r SIZE" is specified, it dumps areas starting from the address SIZE bytes lower than sp. Both commands can specify the context ID of an object as an argument to display the stack information of the object other than the current one.

Syntax
 stack [context]
 dstack [-max max_size] [-r reverse_size] [context]

Example (dstack -r 0x40)

```
[stack info]
 area: 0x8045f788 - 0x80460388, sp: 0x80460238
[stack dump]
804601f8: 00000000 fffffffe 806aeb1c 8046023c  ..........j.<.F.
80460208: 80460218 00000008 806b5990 806aecbc  ..F......Yk...j.
80460218: 00000002 00000000 00000000 806a2384  .............#j.
80460228: 802154c8 80443fc0 806b5990 80651c20  .T!..?D..Yk. .e.
80460238: 00000001 00000000 0000000a 806a2384  .............#j.
...
80460378: 00000002 00000000 80000035 6a7b8c9d  ........5.....{j
```

tlb

This displays the content of the TLB.

Syntax
 tlb

Example

```
[TLB registers]
wired reg = 2

index  Mask       EntryHi    EntryLo0   EntryLo1
******* wired entries *******
 0/48: 0x00000000 0x00002000 0x0000009f 0x000000db
 1/48: 0x00000000 0x00004000 0x0000011b 0x0000015b
***** non wired entries *****
 2/48: 0x00000000 0x80000000 0x00000000 0x00000000
 3/48: 0x00000000 0x80000000 0x00000000 0x00000000
...
47/48: 0x00000000 0x80000000 0x00000000 0x00000000
```

dump

The dump command displays the content of memory with hexadecimal numerals. You must specify an address as an argument. If a size is not specified, the default size 0x100 is used. If the option -w is specified, it displays every 2 bytes. If the option -l is specified, it displays every 4 bytes. On the right side of the display, an ASCII representation is displayed for every byte. If characters cannot be displayed, a period '.' is displayed instead. This character display is not changed by option -l and -w.

Syntax
dump [-w|-l] addr [size]

Example (dump 0x80460238 0x40)

```
80460238: 01 00 00 00 00 00 00 00 0a 00 00 00 84 23 6a 80  .............#j.
80460248: 90 59 6b 80 90 59 6b 80 90 59 6b 80 d4 fe 64 80  .Yk..Yk..Yk...d.
80460258: bc 05 6a 80 00 00 00 00 cc 01 46 80 0a 00 00 00  ..j.......F.....
80460268: 90 59 6b 80 00 00 00 00 00 00 00 00 a0 02 46 80  .Yk...........F.
```

play

This is used to output a sound. The argument specifies a melody.

Syntax
play melody

Description on the argument melody:

exception
This is used to notify that the Exception Monitor starts.

warning
This warns that a Memory Stick is being written to. The melody is automatically looped.

finish
This indicates the termination of the writing to theMemory Stick.

Chapter6 Remote Processing OPEN-R

Remote Processing OPEN-R is a remote processing environment where you can execute an OPEN-R based program on a remote host which is not AIBO.

By using Remote Processing OPEN-R, some objects can be executed on the remote host(connected to AIBO via wireless LAN), and other objects can be executed directly on AIBO. All objects will be executed as one program, distributed between the two machines.

Here are some advantages of Remote Processing OPEN-R:

- Objects executing on the remote host can reconnect to objects executing on AIBO without interrupting AIBO fs execution. So, you can shorten the turn-around time for coding, executing, and debugging, and develop a program efficiently.

- There is source code compatibility between AIBO's objects and remote host's objects. So, you can use rich debugging tools(e.g. gdb) on the remote host.

- While executing a program, you can use the rich resources and various functions that the remote host PC provides.

In Remote Processing OPEN-R, we use TCPGateway objects on AIBO and also on the remote host. TCPGateway is the OPEN-R object that implements communication between the objects executing on AIBO, and the objects executing on the remote host. The communication between AIBO's objects and the remote host's objects are done by passing the ordinary protocol of the OPEN-R inter-object communication over the wireless LAN.

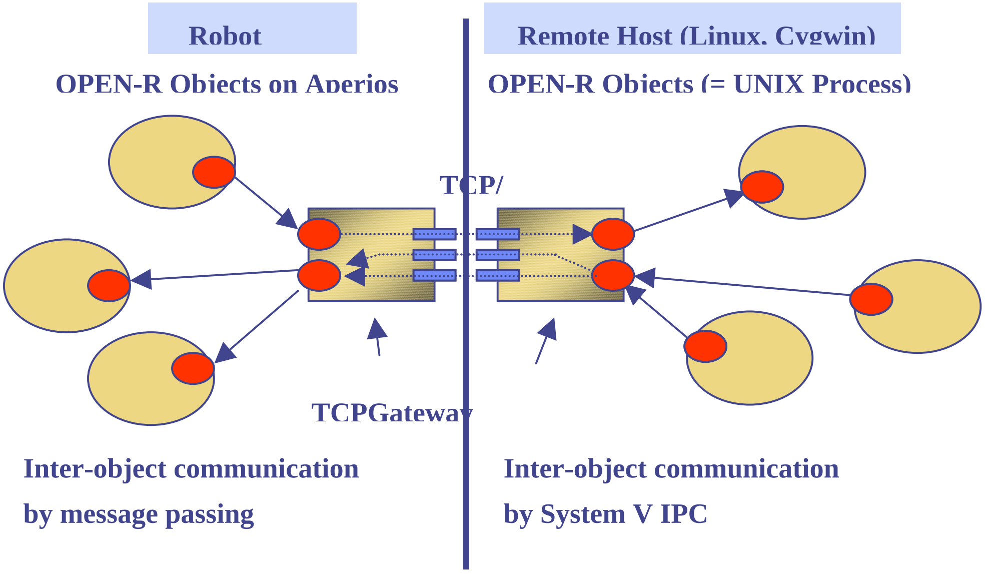

Robot
Remote Host (Linux, Cygwin)
OPEN-R Objects on Aperios
OPEN-R Objects (= UNIX Process)
TCP/
TCPGatewa
Inter-object communication
by message passing
Inter-object communication
by System V IPC

Fig5 Inter-object communication

There is source code compatibility between AIBO's objects and remote host's objects, but there is no binary compatibility.

Remote Processing OPEN-R gives you the environment for building native (x86) binary executable from source code that works on AIBO (by compiling with the OPEN-R SDK). The binary files that work on Remote Processing OPEN-R are not identical to the binary files that work on AIBO.

Chapter7 Safety Guidelines

When you develop programs for AIBO, please take account of the safety of users, as well as the durability of AIBO. Refer to the following web site for details.

http://openr.aibo.com/openr/eng/no_perm/notice.html

Appendix

A. Library Functions

You can use the following functions in your program. The libraries that are automatically linked by the mkbin command provide these functions.

A.1 File system

The following functions are provided for accessing the file system. write() and read() are also used to access the terminal.

open, close, read, write, lseek, fstat, stat, unlink, isatty
mkdir, rmdir, opendir, closedir, readdir, rewinddir

These functions act like the UNIX APIs with the same names. Below are the behaviors specific for the OPEN-R SDK.

- A file descriptor less than 3 represents the terminal device (telnet terminals); There is no difference among the file descriptors 0, 1, or 2.

- File descriptors are global to the entire system; an object can use file descriptors opened by other objects.

- There are the following restrictions regarding file paths:

    - The directory delimiter is "/".

    - A path must begin with "/MS"/. "/MS/" represents the root of the AIBO Programming Memory Stick. A relative path is not allowed.

    - A file name must have 8+3 format. The base name and the extension must be separated by “.”.

      For example
      BASENAME.EXT (correct)
      BASENAMEEXT (incorrect)

    - Usable characters are:

      abcdefghijklmnopqrstuvwxyz
      ABCDEFGHIJKLMNOPQRSTUVWXYZ
      0123456789
      $&#{}~%'()-@^`!_

      Lowercase letters are automatically converted to uppercase letters.

- open() allows the following flags;
  O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREATE, O_TRUNC, O_EXCL

- open() ignores the mode_t argument.

- open() can re-open a file that is already opened only if O_RDONLY is specified for both calls of open().

- isatty() returns 1 if the file descriptor is less than 3. Otherwise, it returns 0.

- Only the following fields in the structure returned by stat() and fstat() are meaningful. Other fields have undefined values.

st_size:	File size. This value may be incorrect if the file is opened.
st_mode:	Permissions are always 0777. S_IFREG bit is 1 for a file. S_IFDIR bit is 1 for a directory.

- mkdir() ignores the mode_t argument.

- The maximum number of entries in a directory that readdir() can read is 682.

- The directory structure returned by readdir() contains a field d_name[], which is a null terminated string of the name of the directory entry, in “BASENAME.EXT” form. Values of other fields in the structure are undefined.

- readdir() does not return directory entries for volume labels and long filename entries.

A.2 C standard library

The OPEN-R SDK uses newlib-1.9.0 for the C library. Below are the behaviors specific for the OPEN-R SDK.

- Functions that depend on the following UNIX APIs do not work:

  execve, fork, getpid, gettimeofday, kill, link, time, wait, fcntl

  The following functions come under this classification.

  clock, fdopen, raise, signal, system, mkstemp, mktemp, tempnam, tmpfile, tmpnam

- Functions that depend on the following UNIX APIs have behavior specific to the OPEN-R SDK, as described in the previous section.

  open, close, read, write, lseek, fstat, stat, unlink, isatty

  The following functions come under this classification.

  fclose, fflush, fgetc, fgetpos, fgets, fileno, fiprintf, fopen, fprintf, fputc, fputs, fread, freopen, fseek, fsetpos, ftell, fwrite, getc, getchar, getopt, gets, getw, iprintf, perror, printf, putc, putchar, puts, putw, remove, rewind, setbuf, setvbuf, ungetc, vfiprintf, vfprintf, vprintf, scanf, fscanf

- The following functions do not use newlib's implementation.

  - exit() stops the execution of the object. The object enters the state for waiting for a message. Destructors for C++ static variables are never called.

  - atexit() does nothing.

  - abort() prints the message that tells that abort() is called. Then it stops the execution of the object. The object enters the state for waiting for a message.

  - rename() can change the path of a file or a directory. See the restrictions about paths above.

  - For memory allocating/deallocating functions such as malloc() and free(), dlmalloc library version 2.7.0 is used instead of 2.6.4 included in newlib.

A.3 C++ standard library

The SDK uses libstdc++-v3 included in the gcc package for the C++ standard library. Because it is built on top of newlib, the description above also applies to C++ functionalities.

B. stub.cc

Object source code must satisfy the following requirements. If you are not using the stub generator, please pay attention to them. (Code generated by the stub generator automatically satisfy these requirements.)

- ❑ **ObjectEntryTable**
  An object must have a structure named `ObjectEntryTable'. It specifies the binding between selector numbers and entry points. ObjectEntryTable has the following form:

```
#include<ObjectEntryTable.h>

ObjectEntry ObjectEntryTable[] = {
   {SELECTOR_NUMBER, (Entry)ENTRY_POINT},
   ...
   {UNDEF, (Entry) ENTRY_UNDEF}
};
```

SELECTOR_NUMBER is a non-negative integer. ENTRY_POINT is a function that is called when the Object received a message whose selector is SELECTOR_NUMBER. ENTRY_POINT must be a function defined using the GEN_ENTRY macro, as described below.

- ❑ **GEN_ENTRY macro**
  The GEN_ENTRY macro, taking two arguments ENTRYNAME and FUNNAME, defines an entry point function with the name ENTRYNAME. The function is a kind of a wrapper function; it sets up some registers and calls the function with the name FUNNAME. All functions registered in ObjectEntryTable must be defined using this macro. The function FUNNAME must have "C" linkage and takes one argument, which is of type void * and points to the data of the received message. The GEN_ENTRY macro is defined in apsys.h.

- ❑ **PrologueEntry**
  The following line must exist in the source code of an object:
  GEN_ENTRY(PrologueEntry, Prologue);

  This defines an entry point function PrologueEntry(), which is called when the object is loaded. PrologueEntry() calls Prologue() defined in libapsys.a, which is automatically linked to an object by the mkbin command. Prologue() does some initialization and calls constructors for global variables of C++.

- **Example**
  Below is an example of code that satisfies the requirements above.

```
#include <ObjectEntryTable.h>
#include <apsys.h>

extern "C"
void method0(void *msg)
{
 ... //  do something using msg
}

extern "C"
void method1(void *msg)
{
 ... //  do something using msg
}


GEN_ENTRY(method0entry, method0);
GEN_ENTRY(method1entry, method1);
GEN_ENTRY(PrologueEntry, Prologue);

ObjectEntry ObjectEntryTable[] = {
   {0, (Entry)method0entry},
   {1, (Entry)method1entry},
   {UNDEF, (Entry) ENTRY_UNDEF}
};
```