CiAO User Guide (Revision: 822)

Wanja Hofer
(LastChangedBy: wanja)

LastChangedDate: 2008-12-22 17:22:03 +0100 (Mon, 22 Dec 2008)

Contents

1 Input to Be Provided by the CiAO Application Programmer

1.1 `config.xml`: Configuration of the OS Objects

This file contains the configuration of the CiAO application in terms of which OS objects it uses (e.g., tasks, ISRs, resources) and how they are allocated to abstract components. See `tools/xml/sample.xml` for a sample configuration file. See `tools/xml/CiAOApp.xsd` for the schema definition of a CiAO application configuration. See `tools/xml/Makefile` for instructions on how to validate a configuration against the CiAO application configuration schema using `xmllint`.

1.2 Hooks Header Files

AUTOSAR hooks have to be defined in separate header files so that they can be inlined; the definitions should include the keyword `inline`. An include guard should guard the whole header; the `as/AS.h` header can be included if system services are to be called from within hook functions (see the AUTOSAR specification for legal system services in hook context).

CiAO allows for the binding of more than one hook function to a particular hook. Hence, if hooks are to be used, `config.xml` has to include:

1. A list of all hooks header files (surrounded by the tag `<hooksheaders>`).
2. The names of all user hook functions as attributes of the hooks that they represent (e.g., `<pretaskhook name="UserPreTaskHook1"/>`).

1.2.1 A Minimal Hooks Header File

```
#ifndef __HOOKS_H__
#define __HOOKS_H__

#include "stdio.h"
#include "as/AS.h"

inline void UserPreTaskHook1 () {
    TaskType id;
    AS::GetTaskID (id);
    printf ("PreTaskHook1 before Task %i.\n", id);
}

#endif // __HOOKS_H__
```

1.3 Component Implementation Files

A CiAO application can consist of an arbitrary number of components and component implementation files. They contain the application code in the form of task definitions, definitions of category 2 ISRs, and definitions of category 1 ISRs. The function definitions have to correspond to the configuration in `config.xml`; that is, both their function names and their component (mapped to a class) have to match the configuration. Necessary includes contain the component headers, the AUTOSAR system services header (`as/AS.h`), and `objects.h`, which contains the generated object definitions.

1.3.1 A Minimal Component Implementation File

```
#include "appcode.h" // component header
#include "as/AS.h"
#include "objects.h"
```

```
#include "stdio.h"

void Alpha::functionTaskTask0 () {
    printf ("Task0 starting...\r\n");
    AS::TerminateTask ();
}
```

1.4 Component Header Files

The component header files bear information on the supplied components. This includes both the declaration of tasks and category 2 ISRs inside a component and the definition of component-local data. The header should have an include guard and include `app/componentheader.h`, which defines macros to facilitate the component description.

Because of problems with the aspect weaver and macros, the declaration has to be both public and manual, `functionTask` is to be prepended to the task name.

1.4.1 A Minimal Component Header File

```
#ifndef __APPCODE_H__
#define __APPCODE_H__

#include "app/componentheader.h"

CIAO_COMPONENT(Alpha)
public:
    static void functionTaskTask0();
};

#endif // __APPCODE_H__
```

1.5 File with User-Defined `main ()`

If the system integrator chooses to write a user-defined `main ()` method, then an additional compilation unit can be supplied that defines `main ()`. Since the CiAO `main ()` implementation has weak linkage, it is then overridden by the one supplied by the integrator. At the end of `main ()`, `AS::StartOS ()` has to be called with the desired application mode.

1.5.1 A Minimal `main.cpp` File

```
int main () {
    AS::StartOS (OSDEFAULTAPPMODE);
}
```

2 Output to Be Provided by the CiAO Generator

2.1 `bind.ah`: Hooks and ISR Binding Aspect File

This file contains the aspect binding the user hook functions to the system-internal hooks. It has to include all user hooks header files so that the hook functions can be inlined. Both the information about which user function is to be bound to which internal hook and the information which user hooks header files to include is provided by `config.xml`. If a particular hook provides an argument, this argument has to be supplied to the user hook function. The pointcuts for the internal hook functions is provided by `as/autosar_pointcuts.ah`, which is to be included accordingly.

2.1.1 A Minimal `bind.ah` File

```
#ifndef __BIND_AH__
#define __BIND_AH__

#include "as/autosar_pointcuts.ah"
#include "hooks.h"

aspect Hooks {
    advice execution (asInternalPreTaskHook ()) : before () {
        UserPreTaskHook1 ();
        UserPreTaskHook2 ();
    }
    advice execution (asInternalShutdownHook ()) : before () {
        UserShutdownHook (* tjp->arg <0> ());
    }
};

#endif // __BIND_AH__
```

2.2 `main.cpp`: OS Object Instantiation and `main ()`

The generated `main.cpp` compilation unit bears

- the definition of memory protection symbols (FIXME: description by Jochen)
- the allocation of task stacks if applicable
- the definition of tasks using the `TASK_INITIALIZER` macro, which is generated by pure::variants to produce only the necessary static initializer fields
- the definition of application modes
- the definition of applications using the `APPLICATION_INITIALIZER` macro, which is generated by pure::variants to produce only the necessary static initializer fields
- the instantiation of the components as configured by the application programmer
- the `main ()` method with default behavior (starting the OS in `OSDEFAULTAPPMODE`), with weak linkage

It includes `../cfInitializers.h` (generated by pure::variants) to provide the macros mentioned above.

2.3 `objects.h`: OS Object Identifiers

This file contains the definitions of the OS object identifiers for all configured objects. The user has to include this header to be able to address an OS object by its (configured) name and not by its internal ID. It has to be have an include guard and itself include `as/autosar_types.h` for the definition of the OS object types.

2.3.1 A Minimal `objects.h` File

```
#ifndef __OBJECTS_H__
#define __OBJECTS_H__

#include "as/autosar_types.ah"

const TaskType Task0 = 0;

#endif // __OBJECTS_H__
```

2.4 `ldciao.x`: CiAO Linker File

FIXME: Description by Jochen

2.5 `ucinfo.h`: Assignment of Components to Applications

FIXME: Description by Jochen

2.6 `app_pointcuts.h`: Generated Application Pointcuts

This file includes the definition of the following pointcuts:

- `asTaskFunctions ()`: The names of all task functions, concatenated by `||`
- `asISR2Functions ()`: The names of all category 2 ISR functions, concatenated by `||`

FIXME: Description by Jochen

3 CiAO System Calls

The CiAO system calls basically correspond to the ones offered by OSEK/AUTOSAR OS. However, since the system calls are all included as static functions in an `AS` class, all calls have to be prepended a `AS::`.

A Hello World for CiAO Newbies

1. SVN access to CiAO repository
   (a) Generate password hash: `htpasswd -n -m newbie`
   (b) Include entry in `/proj/i4ciao/svn/svn-auth-file`
   (c) Generate access entry in `/proj/i4ciao/svn/svn-access-file`
2. Prepare checkout
   (a) Perform checkout: `svn co --username newbie https://www4.informatik.uni-erlangen.de:8088/i4ciao/ciaoos`
   (b) Initialize: `./after-first-checkout.sh`
   (Script copies checked-in *.svn-template files to non-versioned (i.e., ignored) files; good for volatile project and variant description files.)
3. Prepare Eclipse environment

```
Task0 starting...
Task0 activating Task1.
Task0 scheduling.
Task1 starting...
Task1: ID1, state of Task0 2.
Task1 chaining itself...
Task1 starting...
Task1: ID1, state of Task0 2.
Task1 ending...
Task0 ending...
```

Figure 1: Output by the hello world variant.

(a) Open Eclipse: `/proj/i4ciao/tools/bin/eclipse`

(b) Import CiAO: File, Import, General, Existing Projects into Workspace, Root Directory: ciaoos, Select ciao, Uncheck Copy projects into workspace, Finish

(c) Popup Accept Digital Certificate: Accept Permanently

(d) Popup Enter SVN Username and Password: Password, OK

(e) pure::variants licence: Window, Preferences, Variant Management, pure::variants License, Install License, `/proj/i4ciao/tools/pv_licenses/uni-erlangen-2_0.license`

4. Open and generate Hello World variant

   (a) Open variant in Project Explorer: ciao, cs_tc_triboard1796, double-click hello.vdm

   (b) Show pure::variants attributes (some pieces of configuration are defined that way): Right-click any feature, Tree layout, Show all

   (c) Uncheck Variant, Auto Resolve

   (d) Transform model through Variant, Transform or corresponding icon

   ⇒ Popup Transformation successfully finished

   (Transforming means copying the appropriate files to `config/tc_triboard1796` and generating some files according to the selected features.)

   (e) Compile by opening view Make Targets, ciao, config, tc_triboard1796, os, test, ciao, os, krn, Simple_gen, right-click, Add Make Target, all, Create; double-click created target

5. Execute CiAO application

   (a) Log on to Windows server: `rdesktop faui47a`, domain C4

   (b) Open Lauterbach software: Start, Programme, TRACE32, Trace32 ICD Ethernet on Host

   (c) Load image: File, Run Batchfile, `config/tc_triboard1796/os/test/ciao/os/krn/Simple_gen/run.cmm`

   (The batch file prepares the Lauterbach debugger, loads the image, runs, and breaks at `main()`.)

   (d) Start executing: Go

   (e) Meanwhile observe output on PC connected to serial interface: log on to faui49d, `sudo screen /dev/ttyS0`

   (The output should be the one depicted in Figure 1.)