Porting Oracle with ANDF Using an Extended Common API

Thomas J. Watt Jr.

Open Software Foundation
Research Institute

June 1993

Porting a very large existing application to the ANDF technology presents some special challenges. After defining an extended common API, we compiled several major components of the database management system Oracle into platform-independent ANDF, installed them on two important platforms, and tested them successfully using Oracle's QA suite.[1]

1. Introduction

This paper reports on our experience in the OSF Research Institute with using the ANDF compiler technology to build and execute several parts of Oracle.

We assume that the reader is somewhat familiar with Oracle, and so give only cursory information about it in the following section.

1. This work was sponsored by a cooperative research agreement between the Open Software Foundation (OSF) and Unix System Laboratories (USL).

Essentially, we are pursing the goal of increasing the robustness of the ANDF compiler technology toward an industrial strength level of quality such that it delivers product quality for applications across a wide variety of heterogeneous platforms. Toward that end, we have, to date, successfully ported two major and three minor functional parts of Oracle using the ANDF compiler in a native mode. All components of these parts have been compiled, loaded and locally tested within the components directory structure. The latest code drop from DRA has been used in this effort. Oracle is but one of a number of industrial strength applications in our robustness project plan.

We have analyzed the Oracle header files and prototyped an extended common API which successfully abstracts the shape of the source code beyond the purview of the XPG/3 ANDF header file set in a machine and platform independent way. This extended API is sufficient to use the ANDF compiler to distribute intermediate ANDF (`.j`) files which install and properly execute these parts of Oracle between the above two platforms. All of the above mentioned components except one source code file in the **Rdbms** component have been compiled into this extended common API ANDF (`.j` file) format. A temporary workaround utilizes the native mode version of the (`.j`) intermediate file. Further investigation will insure that the API is sufficient across a wider set of platforms. The other components have been subjected to the internal Oracle QA test with successful results.

In subsequent sections we indicate briefly the current status of our porting investigation, introduce the notion of an extended common API approach with the ANDF compiler technology, describe the extended API in some detail, and render preliminary conclusions.

2. Software Information

Software Category

Oracle is the well known Relational Database Management System (RDBMS) product.

Oracle Version and Release Level

Oracle version 6.0.36 source code was used.

Authors and source

The authors of the Oracle RDBMS product are the Oracle Corporation.

Oracle RDBMS represents just under 1.9 million lines of C source code, of which the components mentioned herein represent approximately 500,000 lines of code.

ANDF Technology Version Release

Release TDF-930127 of the ANDF Technology was used; it is based on the TDF Specification Issue 2.0 Revision 1 dated December 1992.

OS Platform Environments

Ultrix 4.2 DECstation 3100 (MIPS) and SCO System V 3.2.1 (i486) with ODT 1.0.

3. Current Status of the Oracle Porting Investigation

The two major functional parts of Oracle which have been ported with the ANDF compiler technology in a native mode are the **Rdbms** and **Forms30**.

The three minor functional parts of Oracle were **Sqlcalc**, **Sqlplus**, and **Report**.

Initially, we attempted to compile using the native compilers on both platforms and were not always successful. On the Ultrix platform we assume this was because we were not attempting a full-blown build of Oracle from scratch and not all parts of Oracle were being built in the usual order including some pieces which are dynamically generated on the fly by various Oracle preprocessors. On the SCO SYSV 3.2 platform, this was, however,

not the case. Oracle source code caused the native compiler (`rcc`) to exceed fixed table limits, and the bundled `cc` compiler was not able to preprocess successfully. It seems that the versions we have, release level 3.2.1 and ODT level 1.0, are not the latest releases of iX86 platforms in use by Oracle for porting. Upgrades to release level 3.2.4 and ODT 2.0 were recommended by Oracle and presumably cure the native compiler (`rcc`) problems.

Once the method for generating Makefiles was suitably modified to use the ANDF compiler, and the define flags were consistent with navigating through the native platform header files on the above platforms, the ANDF compiler sailed through builds of all of the above components (in native mode). All of the minor components have been successfully tested in the same manner used by Oracle Corporation for QA testing prior to shipment of product.

The ANDF Porting experiment for Oracle calls for using the ANDF compiler technology in a strict ANDF header file mode, *i.e.* no native header files are used. Native headers which contain target dependent definitions of C constructs used in compiling Oracle are disallowed.[2]

As stated above, all but one of the **Rdbms** component files have been successfully ported to the ANDF intermediate file format which is both machine and platform independent. For the **Rdbms** component alone, this consists of a total of 596 ANDF intermediate files (`.j`).

We have analyzed the Oracle header files and prototyped an extended common API which is both machine- and platform- independent. For each install site, it is then necessary to build an extended API token definition library which maps into the native platform. At present, Install scripts with the `.j` files for each Oracle component comprise the ANDF distribution. The work which remains for the **Rdbms** component requires only that some unresolved symbol linking problems be resolved on either platform. This appears to be a straightforward porting task which may involve rebuilding some of the libraries which participate in the linking.

2. See “Porting to ANDF” by S. Macrakis, in *ANDF Technology: Collected Papers,* Volume 1, January 1993

4. The Extended Common API Approach

The notion of an extended common API approach to porting application software with the ANDF compiler technology is nothing more than building ANDF tokens which abstract source code fragments that may vary from platform to platform. The extension occurs due to the fact that new header files are scanned prior to the common API on top of which the extension is built, *i.e.* in this case the XPG/3 header file set. The token declarations are usually introduced in header `.h` files as opposed to executable source `.c` files. These new header files reside under a directory named `oracle` which is sibling to the other ANDF common API header file directories. It is not strictly required to introduce new token header files. For instance, the token `BITVEC` is introduced in `defs/orastd.h`. An `oracle.tl` file contains the token definitions and is placed in the API token definition libraries which are distributed with the ANDF technology. The environment file mentioned below contains a pointer to the `oracle.tl` library file.

Upon initial inspection, it was necessary to determine how best to utilize the existing infrastructure of the Oracle Porting Kit in order to expedite our investigation. We determined that it was straight-forward to modify the top-level `prefix.mk` and `librules.mk` files such that when a `Make.file > Makefile` command was given within any component subtree directory, the appropriate ANDF actions would get built into the local source code Makefiles.

The next step was to create a vanilla version of the machine and platform independent environment for ANDF. This consisted of constructing an ANDF environment file, named `oracle`, which resided with the other ANDF technology environment files for the code drop. The trigger to instantiate its use was edited into the `prefix.mk` modifications above such that the compiler `cc` was invoked as `tcc -Yoracle ...`. The `oracle` environment file contains a directory pointer to scan for the extended common API header files that needed to be created, and also turned on the `not_ansi` and `nepc` flags for the ANDF compilation.

What we saw at the beginning of the port were application-dependent header files which hard-coded absolute path names of platform `/usr/include` header files. There are also application dependent versions of these header files in the directories which are searched prior to the ANDF header file set. Since this was a violation of the ANDF rule to avoid using native header files,

the obvious fix was to again modify the `prefix.mk` file in the Oracle root directory to avoid scanning the include directory `standard_def`. In other words, the ANSI standard header files then were picked up via the ANDF header file set in accordance with our ANDF porting guidelines.

It was necessary to provide some `#define` definitions in the required oracle header files, and some in the ANDF environment file.

For the most part, minor source code changes were required to the extent that casts were required for prototyped call interfaces, such as the `strlen` interface.

It should also be noted that several bug reports with regard to limitations of the ANDF compiler technology were submitted due to the Oracle porting investigation.

5. The Environment Files

The environment files created for the ANDF porting investigation were either native or ANDF, each of which were unique for the particular platform due to differences in directory paths on each platform or native flags required.

For example, the contents of the mips native environment file was:

```
+INCL “-I/usr/include”
+FLAG “-not_ansi”
+FLAG “-nepc”
+FLAG “-D__mips”
+FLAG “-Dmips”
+FLAG “-DLANGUAGE_C”
+FLAG “-D__LANGUAGE_C”
+FLAG “-DMIPSEL”
+FLAG “-D__MIPSEL”
+FLAG “-Dunix”
+FLAG “-D__unix”
+FLAG “-Dultrix”
+FLAG “-D__ultrix”
+FLAG “-Y32bit”
```

In contrast, the contents of the i486 native environment file was:

```
+INCL           “-I/usr/include”
+FLAG           “-not_ansi”
+FLAG           “-nepc”
+FLAG           “-DM_I86”
+FLAG           “-DM_I86SM”
+FLAG           “-DM_SDATA”
+FLAG           “-DM_STEXT”
+FLAG           “-DM_I386”
+FLAG           “-DM_XENIX”
+FLAG           “-DM_BITFIELDS”
+FLAG           “-Di386”
+FLAG           “-DM_INTERNAT”
+FLAG           “-Dunix”
+FLAG           “-DM_UNIX”
+FLAG           “-DM_COFF”
+FLAG           “-DM_SYS5”
+FLAG           “-DM_SYSV”
+FLAG           “-DM_SYS3”
+FLAG           “-DM_SYSIII”
+FLAG           “-DM_WORDSWAP”
+FLAG           “-Y32bit”
```

The ANDF environment file for the mips platform was:

```
+INCL “-I/u2/rsandf/feb-93/tdf/include/include/oracle    \
  -I/u2/rsandf/feb-93/tdf/include/include/xpg3”
+LIB  “-loracle -lxpg3”
+FLAG “-DUNIX”
+FLAG “-DANDF”
+FLAG “-DANSI_HEADERS”
```

In contrast, the contents of the i486 ANDF environment file were:

```
+INCL “-I/usr/rsandf/feb-93/tdf/include/include/oracle   \
  -I/usr/rsandf/feb-93/tdf/include/include/xpg3”
+LIB  “-loracle -lxpg3”
+FLAG “-DUNIX”
+FLAG “-DANDF”
+FLAG “-DANSI_HEADERS”
```

6. Token Declarations

The tokens required for the Oracle ANDF port, as explained above, were distributed between either the extended API header files or integrated into the Oracle application header files.

The token declared in each header file were as follows:

Oracle Extended API header files (…/feb93/tdf/include/include/oracle):

math.h

```
#pragma token STRUCT TAG exception #
#pragma token MEMBER int : struct exception : type #     \
  _exception_type_
#pragma token MEMBER char * : struct exception : name #  \
  _exception_name_
#pragma token MEMBER double : struct exception : arg1 #  \
  _exception_arg1_
#pragma token MEMBER double : struct exception : arg2 #  \
  _exception_arg2_
#pragma token MEMBER double : struct exception : retval #\
  _exception_retval_
```

sparams.h

```
#pragma token EXP rvalue : int : SSTPBLCK #
#pragma token EXP rvalue : int : SSTKBLCK #
```

unistd.h

```
#pragma token FUNC char * ( int ) : sbrk #
```

Oracle header files [Extended API] ($SRCHOME/include):

kdi.h

```
#pragma token FUNC int (ktbbh *) : kdidxl #
#pragma token FUNC int (int) : kdidvl #
```

kdo.h

```
#pragma token FUNC int ( int, int) : kdousz #
```

kdr.h

```
#pragma token FUNC int (int) : KDRHSZ #
```

Oracle header files [Extended API] ($SRCHOME/defs):

orastd.h

```
#pragma token FUNC int ( int ) : BITVEC #
```

```
  #pragma token                                                  \
    PROC ( EXP lvalue : bitvec[] :, EXP rvalue : uword : ) \
    EXP rvalue : void :                                          \
    vecclr #
  #pragma token                                                  \
    PROC ( EXP lvalue : bitvec[] :, EXP rvalue : uword : ) \
    EXP rvalue : void :                                          \
    vecset #
  #pragma token                                                  \
    PROC ( EXP lvalue : bitvec[] :,                              \
           EXP lvalue : bitvec[] :,                              \
           EXP rvalue : uword : )                                \
    EXP rvalue : void :                                          \
    veccpy #
  #pragma token FUNC bool ( bitvec vtr[], uword bt ) : vecbit #
  #pragma token FUNC bool ( bitvec vtr[], uword bt ) : vecbis #
  #pragma token FUNC bool ( bitvec vtr[], uword bt ) : vecbic #
```

Oracle header files [Extended API] (`$SRCHOME/sqlcalc/include`):

usdunx.h

```
  #pragma token EXP rvalue: unsigned char : HLPFILE #
  #pragma token EXP rvalue: unsigned char : OVLFILE #
  #pragma token EXP rvalue: unsigned char : MSGFILE #
```

7. Token Definitions

The method used to define the above tokens required the use of the `tld` command which can be invoked with parameters, *e.g.* `-mc -o oracle.tl *.j`, in order to create the `oracle.tl` token definition library.

The ANDF driver, `tcc`, is used with the **`-Fj`** flag and perhaps some **`-I`***`dir`* paths in order to create the `.j` files containing the platform dependent token definitions on each platform. Path names were modified as required for the i486 platform dependent pathname.

The source files used for this purpose were as follows:

kdi.c

```
  #define __BUILDING_TDF_ORACLE_KDI_H
  #ifndef __WRONG_ORACLE_KDI_H
  #pragma implement interface                                    \
    "/u2/oracle/pmax_ul4/include/kdi.h"
```

```
  #undef kdidxl
  #define kdidxl(tbhdr) ktbdxl(tbhdr, 1, sizeof(kdige))
  #undef kdidvl
  #define kdidvl(nitl) ktbdvl(nitl, 1, sizeof(kdige))

  #endif
```

kdo.c

```
  #define __BUILDING_TDF_ORACLE_KDO_H
  #ifndef __WRONG_ORACLE_KDO_H
  #pragma implement interface
    "/u2/oracle/pmax_ul4/include/kdo.h"
  #undef kdousz
  #define kdousz(n, siz)                                        \
    ((sword) ( -((sword)kcbdtl) + ((sword)ktumxr((n)+3)) + \
    ((sword)sizeof(ktbru)) +                                    \
    ((sword)                                                    \
      max(sizeof(kdoi), sizeof(kdom)+(n)*sizeof(kcol))) +  \
    ((sword) (((KDRMAXCO)+(UB1BITS-1))>>3)) +                   \
    ((sword)(((n)+(UB1BITS-1))>>3)) +                           \
    ((sword)(siz)) ))
  #endif
```

kdr.c

```
  #define __BUILDING_TDF_ORACLE_KDR_H
  #ifndef __WRONG_ORACLE_KDR_H
  #pragma implement interface
    "/u2/oracle/pmax_ul4/include/kdr.h"

  #undef KDRHSZ
  #define KDRHSZ(flags) (                                       \
    3*sizeof(ub1) +                                             \
    (bit(flags,KDRHFC) ? sizeof(ub1) : 0) +                     \
    (bit(flags,KDRHFK)                                          \
        ? (sizeof(b2)+sizeof(kd_brid))<<1                       \
        :0)      +                                              \
    ((bit(flags,KDRHFF) ? 1:0) & (!bit(flags,KDRHFH) ? 1:0))\
    ? sizeof(kd_brid)                                           \
     : 0 )  +                                                   \
    (bit(flags,KDRHFL) ? 0 : sizeof(kd_brid)) )
  #endif
```

math.c

```
  #define __BUILDING_TDF_ORACLE_MATH_H
  #ifndef __WRONG_ORACLE_MATH_H
  #pragma implement interface                                   \
    "/u2/rsandf/feb93/tdf/include/include/oracle/math.h"
```

```
  #include “/usr/include/math.h”
  #endif
```

orastd.c

```
  #define __BUILDING_TDF_ORACLE_ORASTD_H
  #ifndef __WRONG_ORACLE_ORASTD_H
  #pragma implement interface  \
    “/u2/oracle/pmax_ul4/defs/orastd.h”

  #define BITVEC(n) (((n)+(UB1BITS-1))>>3)
  void vecclr(bitvec vtr[],uword size);
  #define vecclr(vtr, size)                                  \
    (genclr((ptr_t)(vtr), (size_t)BITVEC(size)))
  void vecset(bitvec vtr[],uword size);
  #define vecset(vtr, size)                                  \
    (DISCARD memset((ptr_t)(vtr), ~((ub1)0),                 \
                    (size_t)BITVEC(size) ) )
  void veccpy(bitvec dest[], bitvec src[], uword size);
  #define veccpy(dest, src, size)                            \
    (DISCARD memcpy((ptr_t)(dest), (ptr_t)(src),             \
                    (size_t)BITVEC(size) ) )
  bool vecbit(bitvec vtr[], uword bt);
  #define vecbit(vtr, bt)                                    \
    ( (vtr)[(bt) >> 3] & ((ub1) (1 << ((bt) & (UB1BITS-1)))))
  bool vecbis(bitvec vtr[], uword bt);
  #define vecbis(vtr, bt)                                    \
    ( (vtr)[(bt) >> 3] |= ((ub1) (1 << ((bt) & (UB1BITS-1)))))
  bool vecbic(bitvec vtr[], uword bt);
  #define vecbic(vtr, bt)                                    \
    ( (vtr)[(bt) >> 3] &= ((ub1)~(1 << ((bt) & (UB1BITS-1)))))
  #endif
```

sparams.c

```
  #define __BUILDING_TDF_ORACLE_SPARAMS_H
  #ifndef __WRONG_ORACLE_SPARAMS_H
  #pragma implement interface                                \
    “/u2/rsandf/feb93/tdf/include/include/oracle/sparams.h”
  #define ULTRIX
    /*use SYSV_386 for SCO_UNIX, ULTRIX for PMAX DECstation*/
  #include “/u2/oracle/pmax_ul4/libs/sosd/sparams.h”
  #endif
```

unistd.c

```
  #define __BUILDING_TDF_ORACLE_UNISTD_H
  #ifndef __WRONG_ORACLE_UNISTD_H
  #pragma implement interface                                \
    “/u2/rsandf/feb93/tdf/include/include/oracle/unistd.h”
```

```
  extern char *sbrk();
  #endif
```

usdunx.c

```
  #define __BUILDING_TDF_ORACLE_USDUNX_H
  #ifndef __WRONG_ORACLE_USDUNX_H
  #pragma implement interface                                    \
    "/u2/oracle/pmax_ul4/sqlcalc/include/usdunx.h"

  #define HLPFILE (unsigned char)                                \
    "/u2/oracle/pmax_ul4/sqlcalc/admin/sqlcalc.hlp"
  #define OVLFILE  (unsigned char)                               \
    "/u2/oracle/pmax_ul4/sqlcalc/admin/sqlcalc.ovl"
  #define MSGFILE  (unsigned char)                               \
    "/u2/oracle/pmax_ul4/sqlcalc/admin/sqlcalc.msg"
  #endif
```

8. The Prefix.mk Modifications

The following differences highlight the changes made to the prefix.mk file in `$SRCHOME`. Subsequent invocation of `Make.file > Makefile` in any one subdirectory with a local .mk file created the desired ANDF modification to the local Makefile.

```
  < MAKE= make $(MKFLAGS)
  > MAKE= make -k $(MKFLAGS)

  < SPFLAGS=-DSYSTEM_FIVE -Y
  > SPFLAGS=-DSYSTEM_FIVE

  < CC=cc
  > CC=tcc -Yoracle -not_ansi -nepc

  < CCFLAGS=-Wf,-XNd9000
  > CCFLAGS=

  < OTHERLIBS= -ldnet -lcV
  > OTHERLIBS= -lcV

  < LIBDNT= $(NETHOME)/dnt/libdnt.a
  > LIBDNT=

  < NETLIBS= $(LIBUTT) $(LIBDNT) $(LIBASYNC) $(LIBTLI)     \
```

```
<   $(LIBLU62) $(LIBX25) $(LIBOSI) $(LIBSQLNET)
> NETLIBS= $(LIBUTT) $(LIBASYNC) $(LIBTLI) $(LIBLU62)    \
>   $(LIBX25) $(LIBOSI) $(LIBSQLNET)

< NETLIBS= $(LIBUTT) $(LIBTCP) $(LIBDNT) $(LIBASYNC)     \
<   $(LIBTLI) $(LIBLU62)
> NETLIBS= $(LIBUTT) $(LIBTCP) $(LIBASYNC) $(LIBTLI)     \
>   $(LIBLU62)

< SRCINCLUDE= $(DEFSINCLUDE) $(CCINCLUDE) $(SOSDINCLUDE) \
<   $(STDINCLUDE)
> SRCINCLUDE= $(DEFSINCLUDE) $(CCINCLUDE) $(SOSDINCLUDE) \
>   #$(STDINCLUDE)
```

The `#$(STDINCLUDE)` on the above line had the desired effect of avoiding the Oracle application's own version of platform dependent standard definitions which were provided via the ANDF header files. Note that `libdnt.a` (`LIBDNT.a`) was stubbed out for this investigation meaning that this version of Oracle was intended not to run over DECnet.

9. Makefile Changes for ANDF

The following example changes were made to each component source subdirectory Makefile. Either a shell script or a `make -k andf` command which descended each component source directory subtree generated the ANDF `.j` intermediate files.

```
< .SUFFIXES: .i .pc .ok .h .hok .d .scr .r .j .k

> .SUFFIXES: .o .j .c
> .c.j:
>       $(CC) -Fj $(CFLAGS) $<
> .j.o:
>       $(CC) -c $(CFLAGS) $<
> OBJSJ=${OBJS:.o=.j}
> andf: $(OBJSJ)
```

10. Source Code Modifications

As mentioned above, the majority of source code modifications comprised the insertion of appropriate casts to mollify the ANDF compiler's requirements.

The creation of the ANDF define (see the ANDF environment file), was necessary in `defs/orastd.h` to separate the inserted token declarations from the original source code. The original source code was then accessible only via the native `cc` compiler or native mode use of the ANDF compiler with an appropriate `#ifndef` ANDF. Some source files required that `defined(ANDF)` be appended to the list of `#if defined(…) ||` … items.

Whenever a declaration like `extern int errno;` occurs in application source code, it is necessary to stub it out or delete it when using the ANDF compiler.

Occasionally, it was necessary to modify a `#include` for a header file, such as `<fcntl.h>` instead of `<sys/file.h>`, or `<math.h>` instead of `/usr/include/math.h`.

Declaration assignments like `byte foo[bar] = "\0";` were modified to `{"\0"};`.

The complete set of source code modifications are to numerous to repeat here, however, the flavor given above is accurate, and indicates that the task of porting Oracle was mostly a straight-forward porting task to ANSI C.

11. Preliminary Conclusions

It is clear that the coding and porting practice of Oracle Corporation is very platform target dependent, but is indeed separable, and thus amenable to a fairly straightforward use of the ANDF compiler technology.

We have demonstrated that it is possible to build and execute 4 of the 5 Oracle components entirely with the ANDF extended common API presented herein. The observed performance of these components was the same as the natively built version.

The fifth component, the **Rdbms**, has also been built except for one source file with the same extended API constructed with the ANDF compiler technology. The one source file includes header files which exhibit a form of coding practice regarding pre-processor macro definitions for which there exist outstanding error reports.

Token definition libraries shipped with this ANDF distribution provide the same resolution as the target dependent code. In effect, the ANDF technology abstracts the shape of the source code to be functionally consistent on all such platforms without any appreciable performance penalty.

A suggested way to improve the Oracle source code to accommodate a higher degree of portability between a wider set of platforms and architectures is to replace target dependent code with ANDF tokens via use of the ANDF technology.

Oracle uses an internal performance test suite for its product verification of the major components. This test suite should be used to subject all of the ANDF built components to the same rigorous testing as Oracle products.