AutoCAD Development System™

Fundamentals of ADS I

Introduction

The AutoCAD Development System (ADS) is the most powerful application programming interface for AutoCAD®. ADS is a open, extensible interface to the AutoCAD drawing editor, designed to address limitations in the power and performance of closed, interpreted AutoCAD programming interfaces such as AutoLISP®.

ADS was introduced in AutoCAD Release 10 for OS/2. Support for ADS is a standard feature of all platform versions of AutoCAD Release 12.

This document is for experienced C language programmers who attend the Autodesk, Inc. Training Department's three-day course about ADS. It is designed to be used in class as a supplement to lecture, demonstration and discussion.

It is composed of an introduction to the basic concepts of ADS application programming within the AutoCAD environment, followed by a linear sequence of exercises in ADS programming. The exercises start with a simple program that prints "Hello, world." to the AutoCAD command prompt, and progress to complex operations such as transforming selection-sets and managing extended entity data. Each exercise is prefaced with an explanation of new concepts and ADS library functions introduced in the exercise.

Sample code is used liberally throughout the document. Is is intended to be used as both illustration of ADS application programming concepts, and the basis for the solution of many of the exercises.

Other documents used in this course describe the ADS development environment used in class, i.e., the AutoCAD version and operating system, and the compiler and linker.

Upon successful completion of this course, the C language programmer will have a sound basis for the development of robust and complex ADS applications for AutoCAD.

This course and its documentation are based on AutoCAD Release 12.

What is ADS?

ADS is an interface.

ADS is an **application programming interface** that allows binary executable programs written in C or C++, compiled with industry-standard compilers, to communicate at run-time with the AutoCAD drawing editor.

ADS is a language binding.

ADS is a language binding that allows development of applications for AutoCAD in the C and C++ languages. These are industry standard languages for large-scale commercial program development. (Both AutoCAD and the ADS library and header files are written primarily in C.)

ADS is an application development system.

ADS is a system for developing powerful and sophisticated vertical-market applications for AutoCAD. The AutoCAD Advanced Modeling Extension® (AME), AutoCAD Render, and the AutoCAD SQL Extension™ (ASE) are all ADS applications, as are the most powerful and sophisticated third-party applications available today for AutoCAD.

ADS is C language code.

ADS is a set of C language library and header files, and associated documentation.

How does ADS differ from AutoLISP?

ADS is not a replacement for AutoLISP. AutoLISP is an easy to learn, easy to use macro language that encourages customization of the AutoCAD drawing editor by the non-programmer. ADS is a sophisticated application development environment for the professional programmer.

ADS enhances and works with AutoLISP. ADS applications can extend the AutoLISP interface by offering their own APIs in AutoLISP, and by defining new AutoLISP functions for customer use.

Both ADS and AutoLISP have their own strengths and weaknesses.

AutoLISP Strengths and Weaknesses

AutoLISP is easy to learn and use. Program file size is typically small. It requires no extra-cost develoment tools apart from an ASCII text editor. Its source code is portable on all AutoCAD platforms at the same revision level. It is widely supported by third-party books, tutorials and training classes.

On the other hand, AutoCAD has no binary or random file I/O. It supports only sequential ASCII file access. Since it is an interpreted language. AutoLISP programs exhibit much slower computational performance for numeric operations than compiled ADS executables. Finally, the language is closed to AutoCAD. The AutoLISP development environment consists only of the functions provided by Autodesk. However, it can be extended by third parties through ADS.

ADS Strengths and Weaknesses

ADS uses widely accepted languages for commercial software development. It uses industry standard compilers, linkers and debuggers. It can use commercial C and C++ libraries and classes.

ADS executables exhibit excellent computational performance for numeric-intensive operations. ADS applications are open to all the functionality supported by the operating system, through standard C and C++ language libraries; in other words, an ADS application can be of virtually any size, sophistication and complexity.

Since ADS applications are compiled programs, source code is protected.

On the other hand, ADS application development requires the extra expense of a third-party compiler and linker for each target AutoCAD platform. ADS application source is portable, but binaries aren't.

C and C++ are harder to learn and use than Lisp. The ADS environment is basically a compile-link-debug cycle, rather than an interactive, interpreted environment.

Finally, binary file sizes are typically larger than comparable AutoLISP programs, and ADS library overhead is required in each separate binary.

When Might You Use AutoLISP Rather Than ADS, Or Vice-Versa?

The application developer is the best person to determine whether an application should be written in AutoLISP or ADS. Here are some suggestions that might be of some assistance.

AutoLISP is an excellent environment for prototype programs that will eventually become ADS applications; for applications that do not require binary, random or very large file I/O; and for applications that primarily make AutoCAD command calls.

ADS is an excellent environment for computationally-intensive applications such as AME; for applications that need to take advantage of system level resources; for applications that must do binary, random or very large file I/O;

and for re-use of existing C language application code in a port to an ADS application.

How Does ADS Work?

AutoCAD sends and receives information to and from AutoLISP through a dedicated communications buffer. An ADS application sends and receives information to and from AutoCAD by utilizing this AutoLISP communications facility. (The implementation of the communication buffer is platform dependent.)

An ADS application is a C or C++ language executable program composed of one or more external functions that are loaded by and called from the AutoLISP interpreter. The application includes calls to ADS library functions that initialize communication with AutoLISP and AutoCAD. After the initilization of this communication channel, the ADS application effectively serves as a slave to AutoCAD and AutoLISP. It waits in an infinite *dispatch loop* for *request codes* from AutoCAD. A switch statement within the dispatch loop calls the appropriate action within the ADS application based on the result code it receives from AutoCAD.

What Do I Need To Create An ADS Application?

To create an ADS application, a developer needs the ADS library and header files that ship with AutoCAD in the *ads* directory of a standard installation, and a C or C++ compiler and linker supported for ADS development on the target AutoCAD platform.

Compiler and linker support varies by platform. See the *readme.ads* file in the *acad* directory of a standard AutoCAD installation for a list of the supported compilers and linkers for the platform in question.

Documentation about the supported compilers and linkers is in the *ads/docs* directory. For example, the document *ads/docs/realmode.txt* in a standard Release 12 DOS 386 installation describes support for real mode ADS development using Borland® Turbo C® or Microsoft® C.

ADS Data Types

ADS applications use the **standard C data types** of short, int, long, double, struct, char, etc. In addition, special **ADS-specific data types** are defined by `typedef` statements in the ADS header file *ads.h* to represent AutoCAD floating point numbers, points, selection-sets and entity names, binary data, entities, table records, and system variables.

ADS Specific Data Types

Floating point numbers

```
typedef double ads_real;
```

`ads_real` corresponds to the definition of a double-precision floating point number on the target platform. AutoCAD and AutoLISP always use double-precision to represent floating point values. For convenience, define all double-precision floating point values in an ADS application to be of type `ads_real`.

Points

```
typedef ads_real ads_point[3];
```

A point corresponds to an array of three `ads_real` numbers.

If variable `pt` is of type `ads_point`

```
ads_point pt;

pt[0] = 1.0;
pt[1] = 2.0;
pt[2] = 3.0;
```

then the *X* value of `pt` is `1.0`, the *Y* value of `pt` is `2.0` and the *Z* value of `pt` is `3.0`.

Symbolic codes for members of the array `ads_point` are defined for convenience's sake in the ADS header file *ads.h*.

```
#define X 0
#define Y 1
#define Z 2

ads_point pt;

pt[X] = 1.0;
pt[Y] = 2.0;
pt[Z] = 3.0;
```

Selection-set and entity names

```
typedef long ads_name[2];
```

These are stored as arrays of two long integers.

ADS Specific Structures

Result Buffer

`struct resbuf`

A `resbuf` structure is used to represent entity and table data. It is composed of the following members.

1. A pointer to another `resbuf` structure (used to implement linked lists).
2. A `short integer` declaring the type of data contained within the `resbuf`.
3. A `union` containing one value for any of a number of different data types.

This is the definition of the `resbuf` structure, from *ads.h*.

```
union ads_u_val {
    ads_real rreal;
    ads_real rpoint[3];
    short rint;
    char *rstring;
    long rlname[2];
    long rlong;
    struct ads_binary rbinary;
};

struct resbuf {
    struct resbuf *rbnext;
    short restype;
    union ads_u_val resval;
};
```

Result buffer structures are chained together as linked lists to represent AutoCAD entity and table (or named object) records.

Binary Chunk

`struct ads_binary`

An `ads_binary` structure is used to store arbitary chunks of binary data. It is composed of the following members.

1. A `short integer` describing the length of the data in bytes.
2. A pointer to `char` that represents the binary data.

This is the definition of the `ads_binary` structure, from *ads.h*.

```
struct ads_binary {
    short clen;
    char *buf;
};
```

This structure is used only in extended entity data.

The maximum length of binary data is 127 bytes per structure instance.

ADS Functions

ADS Function Arguments

Unlike AutoLISP functions, all arguments to ADS functions must always be supplied.

- There is no "varargs" approach to ADS function calls.
- ADS library functions were implemented this way to reduce overhead in the ADS library.
- NULL is typically used to represent an optional argument of no value.

The functions `ads_command()` and `ads_cmd()` are special cases.

- They take a variable number of arguments.
- Each argument consists of a pair of items: a keyword describing the data type of the argument, and the value of the argument.
- End with a NULL argument.

For example, this line of code calls the function `ads_command()` and passes to it 9 arguments. 8 of the arguments draw a Line entity between 1,1 and 5,5. The `NULL` argument terminates the argument list.

```
ads_point pt1 = {1.0, 1.0, 0.0}, pt2 = {5.0, 5.0, 0.0};

ads_command(RTSTR, "._LINE", RT3DPOINT, pt1, RT3DPOINT, pt2,
            RTSTR, "", NULL);
```

In another example, this line of code calls the function `ads_command()` and passes to it 7 arguments. 6 of the arguments draw a Circle entity between at 5,5 with a radius of 1 unit. The `NULL` argument terminates the argument list.

```
int rad = 1.0;
ads_point cen = {5.0, 5.0, 0.0};

ads_command(RTSTR, "._CIRCLE", RT3DPOINT, cen, RTREAL, rad, "", NULL);
```

ADS Function Names

The ADS library defines a number of functions. The name of each function is prefixed with "ads_".

Caution Never define a new function in your application with a name that begins "ads". This prefix is reserved for current and future ADS functions, both those that are exported and those maintained internal to AutoCAD.

Counterparts To AutoLISP

ADS functions are essentially equivalent in argument lists and functionality to their AutoLISP counterparts of the same names. Here are some examples of similarity between ADS and AutoLISP function names.

ADS function	AutoLISP function
ads_command();	(command)
ads_getpoint();	(getpoint)
ads_initget();	(initget)

Table 1. ADS functions and AutoLISP counterparts

ADS Function Return Values

Most functions return an integer indicating success or failure, known as a *result code*. Result codes are defined in *adscodes.h*. An ADS library function can return one of six possible result codes: In the table that follows, the most common result codes are listed at the top.

Result code	Meaning
RTNORM	- function completed successfully
RTERROR	- function failed to complete successfully
RTCAN	- user cancelled function, e.g., ads_getpoint()
RTREJ	- AutoCAD rejected the operation
RTFAIL	- the comm link with AutoLISP has failed
RTKWORD	- user entered valid keyword

Table 2. ADS function result codes

An ADS application should **always test the return value of an ADS function** that returns a result code indicating success or failure.

In the examples that follow, we assume the developer has assigned meaningful values to the symbolic codes `BAD` and `USER_CANCEL`. These are not defined by ADS.

In the first example, if `ads_getpoint()` does not return a result code of `RTNORM`, the symbolic code `BAD` is returned.

```
ads_point pt1;

if (ads_getpoint(NULL, "\nPoint: ", pt1) != RTNORM)
   return BAD;
```

In the second example, the variable `stat` is set to the return value of `ads_getpoint()`. If `stat` is equal to `RTCAN`, then the user pressed Ctrl-C during the `ads_getpoint()` prompt. If it's equal to `RTERROR`, something failed in the function call. Otherwise, we assume `ads_getpoint()` returned `RTNORM`. (In some circumstances, `ads_getpoint()` can return `RTNONE` or `RTKWORD`, but not in the case illustrated here.)

```
ads_point pt1;
short stat;

stat = ads_getpoint(NULL, "\nPoint: ", pt1);

switch(stat) {
case RTCAN:
    ads_printf("User cancelled.");
    return USER_CANCEL;
case RTERROR:
    ads_fail("Could not get point from user.");
    return BAD;
default:
    break;
}
```

ADS Program Initialization

An ADS application contains a `main()` function. `main()` uses ADS library function calls to initialize communication with AutoLISP, and an infinite loop called the *dispatch loop*.

After communication with AutoLISP is established, the ADS application responds to various *request codes* from AutoLISP. These request codes are normally dealt with by a `switch` statement within the dispatch loop. At the top of the dispatch loop, the ADS application receives a request code; branchs within the `switch` statement to handle the code appropriately; returns a *result code* to AutoLISP; then, goes back to the top of the dispatch loop and awaits the next request code from AutoLISP.

This is an example of a `main()` function for an ADS application. It defines two function names for AutoLISP to which it will respond: `C:FOO` and `C:GOO`.

```
void
main(argc, argv)
  int argc;
  char *argv[];
{
    int stat;
    short scode = RSRSLT;             /* This is the default result code */

    ads_init(argc, argv);             /* Initialize the interface */

    for ( ;; ) {                      /* Note loop conditions */

        if ((stat = ads_link(scode)) < 0) {

            printf("TEMPLATE: bad status from ads_link() = %d\n", stat);

            /* Can't use ads_printf to display
               this message, because the link failed */
            fflush(stdout);
            exit(1);
        }

        scode = RSRSLT;               /* Default return value */

        /* Check for the following cases here */
        switch (stat) {

        case RQXLOAD:                 /* Register your ADS external functions.
                                         Register your function handlers if you
                                         want your ADS functions to be called
                                         transparent to this dispatch loop.
                                         Required for all applications.  */

            if (ads_defun("C:FOO", 0) != RTNORM) {
                scode = RSERR;
            }
            if (ads_defun("C:GOO", 1) != RTNORM) {
                scode = RSERR;
            }
            break;

        case RQSUBR:                  /* This case is normally expanded to
                                         select one of the application's
                                         external functions */
            switch (ads_getfuncode()) {
            case 0:
                if (foo() != RTNORM) {
                    scode = RSERR;
                }
                break;
            case 1:
                if (goo() != RTNORM) {
                    scode = RSERR;
                }
                break;
            default:
                ads_printf("\nError - no such function defined.");
                scode = RSERR;
                break;
            }

            break;

        case RQXUNLD:                 /* Do C program cleanup here.
                                         Not required unless you need to
                                         clean up your own allocated resources.
```

```
                                        Note: You don't have to undefine ADS
                                        functions.  LISP does it for you.  */
            break;

        case RQSAVE:                 /* AutoCAD SAVE command notification.
                                        You can use it for your own database
                                        synchronization.  Not required.  */
            break;

        case RQQUIT:                 /* AutoCAD QUIT command notification.
                                        Not required.  */
            break;

        case RQEND:                  /* AutoCAD END command notification.
                                        Not required.  */
            break;

        default:
            break;
        }
    }
}
```

This is the standard sequence of events within the `main()` function of an ADS application.

1. Load the application.

AutoLISP loads the application through an `(xload)` or `ads_xload()` function call. The application begins executing; that is, the program becomes a process.

2. Initialize communication with AutoLISP.

The ADS library function `ads_init()` is called once by the application.

3. Tell AutoLISP we're ready for requests.

At the top of the dispatch loop, the ADS library function `ads_link()` is called with an application result code of `RSRSLT`. This tells AutoLISP that the ADS application is ready to receive request codes from the return value of `ads_link()`.

4. Respond to the request to register functions with AutoLISP.

The first time through the dispatch loop, AutoLISP returns from `ads_link()` with a request code of `RQXLOAD`. This tells the ADS application to register its external functions with AutoLISP.

5. Register the external functions.

Every function the application wishes to register is *defined* for AutoLISP by calling the ADS library function `ads_defun()` once for each function. The first argument to `ads_defun()` is the name to define for AutoLISP; the second is a unique integer that AutoLISP will subsequently use to indicate which of the defined functions was requested by the user .

6. Go back to the top of the dispatch loop and wait for a new request code from AutoLISP.

The application calls `ads_link()` again with a result code of `RSRSLT`. At this point, the ADS application is a slave of AutoLISP, waiting for AutoLISP to return from the call to `ads_link()` with a request code that tells it what to do, e.g., call one of its internal functions.

7. Execute on demand one or more of the functions the application defined for AutoLISP.

When `ads_link()` next returns with an `RQSUBR` request code from AutoLISP, the ADS application determines which of the functions it defined for AutoLISP it is being asked to execute. It gets an integer code (matching one it previously defined for AutoLISP) by calling the ADS library function `ads_getfuncode()`, and then executes the appropriate function call.

8. Indicate to AutoLISP the success or failure of the operation.

The ADS application returns to the top of the dispatch loop and calls `ads_link()` with a result code of `RSRSLT` if the requested operation was successful, or `RSERR` if not.

The ADS Library and Header Files

Libraries

Autodesk supplies an ADS function library file for each compiler environment for each platform. The appropriate library file must be linked in to every ADS application. The source for this object library is not provided.

The ADS object library contains the compiled versions of all the ADS functions listed in the *ADS Programmer's Reference*.

Headers

Each ADS application much include three header files:

- *ads.h*
- *adscodes.h*
- *adslib.h*

For convenience's sake, *adslib.h* contains references to include *adscodes.h* and *ads.h*.

Contents

ads.h

1. Type definitions for ADS-specific data types.
2. Prototypes for ADS library functions.
3. Definitions for ads_initget() bit codes.
4. Miscellaneous utility definitions.

adscodes.h

1. Definitions for request, result and result type codes.

adslib.h

1. System-dependent definitions.
2. Miscellaneous utility definitions.
3. Macro to expand _(()) syntax found in function prototype declarations.

How To Create A Simple ADS Program

In this section, you'll learn how about ADS functions, request codes and result codes. This will prepare you to write a simple ADS application, using a template that contains the `main()` function and dispatch loop discussed earlier in this document.

Your program will define an external function for AutoLISP that prints the message "Hello, world." to the AutoCAD command prompt. You'll implement this program in the exercise at the end of this section.

Function Implementations

Three new functions are required for this section.

1. `ads_defun()`
2. `ads_getfuncode()`
3. `ads_printf()`

Defining External Functions for AutoLISP with ads_defun()

```
int ads_defun(const char *sname, short funcno)
```

The ADS library function `ads_defun()` takes two arguments: a character string and a `short` integer.

It registers the character string with AutoLISP, creating a new **external function** of the AutoLISP data type **exsubr**. The integer code must be unique for each function, and is subsequently used by AutoLISP to indicate that the user has called the external function.

`ads_defun()` returns `RTNORM` if it succeeds; otherwise, it returns an error code such as `RTERROR`.

`ads_defun()` is called within the `RQXLOAD` case of the dispatch loop's `main()` function. Each function defined by the ADS application for export to AutoLISP must be registered by a call to `ads_defun()`, and each function must use a unique integer code for the second argument.

In the example that follows, the external function C:FOO is registered with AutoLISP, using the unique integer code `0`.

```
case RQXLOAD:
    if (ads_defun("C:FOO", 0) != RTNORM) {
        scode = RSERR;
    }
    break;
```

Finding The External Function Code with ads_getfuncode

```
int ads_getfuncode(void)
```

The ADS library function `ads_getfuncode()` returns the unique integer code for the external function called by the user. The integer code must have been previously defined for AutoLISP in a call to `ads_defun()`.

`ads_getfuncode()` returns `RTERROR` on failure.

`ads_getfuncode()` is called within the `RQSUBR` case of the dispatch loop's `main()` function. When an ADS application receives an `RQSUBR` request, the user has entered the named of an external function defined for AutoLISP by the ADS application. AutoLISP calls the ADS application and asks it to evaluate the function whose integer code is retrieved by `ads_getfuncode()`.

In the example below, the function `foo()` is called if `ads_getfuncode()` returns the integer code `0`.

```
case RQSUBR:
    switch (ads_getfuncode()) {
    case 0:
        if (foo() != RTNORM) {
            scode = RSERR;
        }
        break;
    default:
        ads_printf("\nError - no such function defined.");
        scode = RSERR;
        break;
    }
```

Printing To The Command Prompt with ads_printf

```
int ads_printf(const char *format, ...)
```

The ADS library function `ads_printf()` prints a character string to the AutoCAD command prompt. It takes the same format string arguments used by the `printf()` function in the standard C library.

`ads_printf()` returns `RTNORM` on success; otherwise, it returns an error code.

The maximum length of the string printed by `ads_printf()` must not exceed 132 characters.

In the example that follows, the character string "What's up, Doc?" is printed to the AutoCAD command prompt.

```
if (ads_printf("\nWhat's up, Doc?") != RTNORM) {
    ads_fail("Error printing message.");
    return RTERROR;
}
```

Request Code Implementations

Two new request codes are required for this section.

1. `RQXLOAD`
2. `RQSUBR`

Within the dispatch loop of an ADS application's `main()` function, calls to `ads_link()` return **request codes** from AutoLISP. The ADS application branches appropriately within a `switch` statement, depending on the request code it received from AutoLISP.

The `RQXLOAD` request code indicates that AutoLISP would like the ADS application to register all of its external functions; so, this is where the application would make all of its calls to `ads_defun()`.

The `RQSUBR` request code indicates that the user has typed in one of the external functions registered by the ADS application. The application makes a call to `ads_getfuncode()` to determine which function name has been typed in by the user (by retrieving the unique interger code registered along with the function name), and then calls the appropriate internal function.

Result Code Implementations

Four new result codes are required for this section.

1. `RSRSLT`
2. `RSERR`
3. `RTNORM`
4. `RTERROR`

An ADS application passes result codes as arguments to `ads_link()` to indicate its status to AutoLISP. `RSRSLT` tells AutoLISP that the last operation was successful, and `RSERR` tells AutoLISP that the last operation failed. The latter is typically used to indicate that the execution of an external function requested by the user failed for some reason.

Most ADS library functions return result codes indicating their success or failure. `RTNORM` indicates success, and `RTERROR` indicates failure. When a function returns `RTERROR`, it typically sets the AutoCAD system variable ERRNO to an integer code indicating the reason for the failure.

EXERCISE

In this exercise, you'll write a small ADS application that prints the string `"Hello, world."` to the command prompt area in AutoCAD.

Objectives

- Implement a simple ADS application.
- Use a template for the ADS `main()` function and dispatch loop.
- Use the ADS library functions `ads_defun()`, `ads_getfuncode()` and `ads_printf()`.

Instructions

When the ADS process receives an `RQXLOAD` request from AutoLISP, define a function named hello for AutoLISP by calling the ADS library function `ads_defun()`. Use an integer code of `0`.

When the ADS process receives an `RQSUBR` request from AutoLISP, get the integer code for the request by calling `ads_getfuncode()`. If the value is `0`, call the function `print_hello()`.

1. Open the (project) file *hello*.
2. Add code to the file *hello.c* that implements the function `print_hello()`.

```
int
print_hello()
{
    if (ads_printf("\nHello, world.") != RTNORM) {
        ads_fail("Error printing message.");
        return RTERROR;
    }
    return RTNORM;
}
```

3. Add code to *hello.c* that defines the prototype for the function `print_hello()`.

```
int print_hello(void);
```

4. Add code to *hello.c* that registers the function `print_hello()` with AutoLISP. Use the ADS library function `ads_defun()` within the `RQXLOAD` case of the dispatch loop's `switch` statement.

```
case RQXLOAD:
    if (ads_defun("print_hello", 0) != RTNORM) {
        scode = RSERR;
    }
    break;
```

5. Add code to *hello.c* that calls the function `print_hello()` if `ads_getfuncode()` retrieves an integer code of `0`. Place this code within the `RQSUBR` case in the dispatch loop's `switch` statement.

```
case RQSUBR:
    switch (ads_getfuncode()) {
    case 0:
        if (print_hello() != RTNORM) {
            scode = RSERR;
        }
        break;
    default:
        ads_printf("\nError - no such function defined.");
        scode = RSERR;
        break;
```

6. Compile and link the program *hello*.
7. Load the program into the AutoCAD editor with the AutoLISP `(xload)` function and call the new external function (`print_hello`).

Command: (xload "hello")

Command: (print_hello)

Debugging An ADS Application

The ADS libraries support interactive debugging using industry-standard debuggers.

Features and Benefits

The application of powerful, robust debugging tools within the ADS development environment allows the programmer to rapidly find and fix bugs within ADS programs.

Objectives

- Review operation of the classroom debugging environment.
- Debug a simple ADS program into which a bug has been introduced.

Each platform and debugger require a different set of steps to create an ADS binary with symbolic debugging information, and load and debug the application within the AutoCAD editor. Your instructor has a paper that describes the debugging environment you will use in this class. For environments found outside of class, refer to the platform and compiler-specific ADS documentation that ships with each version of AutoCAD, and to th documentation that came with your compiler and debugger.

At this time, please review the separate paper on the classroom debugging environment. Do so before beginning the next exericse.

EXERCISE

In the exercise that follows, you'll debug a simple ADS application into which a deliberate error has been introduced.

The ADS application is designed to print a string to the AutoCAD command prompt. However, all that it prints is the null string. Your job is to load the application into AutoCAD and the debugging environment and find the bug.

Objectives

- Compile an ADS application with symbolic debugging information.
- Start the application in the debugger.
- Load the application into AutoCAD and call the external function.
- Use the debugger to find the program's bug.

Instructions

1. Open the (project) *helloerr*.
2. Open the source file *helloerr.c*.
3. Check the compiler options and turn on the "debug" option. Build the program.
4. Set two break points in *helloerr.c*: one at the `ads_link()` call in the `main()` function's dispatch loop, and the other at the call to `strcpy()` in the function `helloerr()`.
5. Start the debugger.
6. Load the program file *helloerr* into AutoCAD with the AutoLISP `(xload)` function. The debugger should bring you to the first break point in the source.
7. In the "Watch" window of your debugger, set up to check the value of the variable `str`.
8. Step through the source until you can't go any further. At this point, the appplication is stopped at the `ads_link()` call waiting for a request code from AutoLISP.
9. In the AutoCAD editor, enter the command HELLOERR. The debugger should become active.
10. Step through the source for the function `helloerr()`. Check the value of the character string `str` to make sure that its set to the appropriate string by the `strcpy()` function.

11. If you haven't found the bug at this point, continue to step through the code. If the ADS application crashes, try to stop the debugger.

TIP • When an ADS application crashes in the debugger, a subsequent attempt to load the ADS application into AutoCAD may lock the AutoCAD drawing editor. Terminate the existing AutoCAD process and launch a new one before continuing. Reload the application into the debugger and AutoCAD.

Review

The bug in this program is in the declaration of the variable `str`. As a pointer to `char`, `str` has no space allocated to it; so, the call to `strcpy()` fails to copy the string as you might expect it to.

You can fix the bug by changing the declaration of `str` to:

```
char str[20];
```

Rebuild the program and check to see if this change fixes the bug.

Calling AutoCAD Commands

ADS applications can make calls to the native AutoCAD command set through the ADS library functions `ads_command()` and `ads_cmd()`. Any command can be called, and any prompt sequence of any complexity can be passed from an ADS application to AutoCAD.

Features and Benefits

The ability to use the existing AutoCAD command set is a fundamental feature of ADS application development. Many ADS applications take advantage of the power and diversity offered by AutoCAD's native command set, using combinations of existing commands to create new ones defined by the application. Sophisticated applications can use other methods to create and modify entities apart from AutoCAD command calls, but they are a basic building block for beginning application development, and required for modification of AutoCAD tables or **named objects** such as Layers.

Objectives

- Learn how to call AutoCAD commands from ADS.
- Learn the syntax of the ADS library function `ads_command()`.
- Use `ads_command()` within an ADS application.

How To Call AutoCAD Commands From ADS

AutoCAD commands can be called from ADS by passing a variable number of paired arguments to the function `ads_command()`. These paired arguments correspond to the command name or one of the command's options, preceded by a symbolic code indicating the data type of the next argument.

The ADS header file *adscodes.h* defines a set of **result type codes** for functions like `ads_command()`. These result type codes are used to indicate the data type of a result value or argument, both to AutoCAD and to the ADS application.

At this time, please refer to your *ADS Programmer's Reference* for the table of result type codes defined by *adscodes.h*. You will use this table extensively in class, so please mark its location (ask your instructor for a Post-It™ Tape Flag or a similar marker).

Function Implementations

One new function is required for this section.

- `ads_command()`

Calling AutoCAD Commands with ads_command()

```
int ads_command(const char*, ...)
```

The ADS library function `ads_command()` takes a variable number of paired arguments. Each pair consists of a result type code and the AutoCAD command or option data.

An `ads_command()` argument list must be terminated by a single argument of either `0` or `RTNONE`.

The special symbolic code `PAUSE` may be used whenever the application needs to allow the user to respond directly to a command's prompt. This code's result type code is `RTSTR`.

`ads_command()` returns `RTNORM` if it succeeds; otherwise, it returns an error code `RTERROR` or `RTREJ`, or `RTCAN` if the user cancels the command during interactive operation.

In this example, the AutoCAD LINE command is issued, and a Line entity is drawn from 1,1 to 5,5 in the current UCS.

```
ads_command(RTSTR, "LINE", RTSTR, "1,1",
            RTSTR, "5,5", RTSTR, "", RTNONE);
```

With error checking added, the example might look like this.

```
int stat;

stat = ads_command(RTSTR, "LINE", RTSTR, "1,1",
                   RTSTR, "5,5", RTSTR, "", RTNONE);

switch (stat) {
case RTCAN:
    return RTNORM;
    break;
case RTERROR:
case RTREJ:
    ads_fail("Error drawing line.");
    return RTERROR;
    break;
default:
    break;
}
```

When To Use...

AutoCAD entities can be created, modified and deleted directly through advanced ADS library function calls discussed later in this course. AutoCAD system variables can be read and written to without using the `ads_command()` function. Nevertheless, for application prototyping and for simple, non-transparent application-defined commands, it is perfectly acceptable to call native AutoCAD commands from within an ADS application.

AutoCAD named objects such as Layers and Views can only be created through AutoCAD command calls. There is no direct ADS library interface that allows an ADS application to create a new layer, for example, without calling an AutoCAD command.

TIP • Most ADS library functions return an integer value indicating success or failure. RTNORM is the result code normally returned by an ADS library function on success. A different result code will be returned on failure, usually RTERROR. **Always** check the return value of an ADS library function call that returns a status code.

Communication With AutoLISP

The ADS library provides a variety of functions that permit an ADS application to return various types of values to AutoLISP.

When a user calls an ADS application by entering a function name that the application has defined for AutoLISP (through ads_defun()), the function will return a value. By default, it will return the AutoLISP symbol nil.

The ads_retXXX() functions allow an ADS function to return to AutoLISP a single value from one of a number of different data types. For example, ads_retint() returns an integer to AutoLISP, ads_retreal() returns a real number, ads_retstr() returns a string, etc

EXERCISE

In this exercise, you'll write a new ADS file Xline.C that contains a command for making a line. Later you will extend it to check the error value. Later you will propogate the return values all the way back to the ads_link() call to AutoLISP; in other words, tell AutoLISP whether we succeeded or failed.

Objectives

- Use new commands **ads_command(), ads_fail()** and **ads_retvoid()**

Instructions

1. Copy the template or hello.c to xline.c

2. Define a new command name for AutoLISP by calling ads_defun() with a string prefaced with "C:".

   ads_defun("C:XLINE", 0);

3. Write a new function called xline(). Use the ads_command() function from the ADS library to call the native AutoCAD command from within xline().

```
int xline()
{
    ads_command(RTSTR, "_.LINE", RTSTR, "0,0",
        RTSTR, "5,5", RTNONE);
    return RTNORM;
}
```

4. Compile and execute xline.c

5. Add error checking through **ads_fail()**

6. If xline() returns RTERROR, set scode to RSERR in the main() function's dispatch loop.

7. Return a blank value from the ADS process to AutoLISP using **ads_retvoid()**

   Add a call to ads_retvoid() immediately prior to call to the keyword *return* within the ADS function.