Object Oriented Programming

by David Miller

If a builder built buildings the way programmers write programs, the first woodpecker would destroy civilization.

-- Anonymous

Introduction

Object-Oriented Programming (OOP). It has easily become one of the most popular buzzwords in the software community in last few years. There are courses on OOP, OO Analysis (OOA), and OO Design (OOD); dozens of books, magazines, and seminars devoted to OO.

What's the fascination?

OOP offers a means of building reusable generic software components.

Imagine what the computer hardware industry would be like if every manufacturer had to make all of their own integrated circuits. In a very real sense, this is the state of much of the software industry today.

How many times have you had to implement a linked list? Or a binary tree? And how many times have you had to stop and draw a diagram to figure out which order to change the references in a doubly linked list to not lose a pointer?

Admittedly, AmigaDOS actually does a better job of managing generic lists than most of its contemporaries. But how do you handle a list of lists? Or a data structure that is really part of more than one list? Perhaps a data structure that describes beef stew and appears on both the animal and vegetable list?

Quick Overview of Object-Oriented Programming

OOP introduces several techniques for dealing with these issues.

One of these is *encapsulation*. Encapsulation is the term used to describe the grouping a set of related data items and providing a well defined interface to the group.

Along with encapsulation, goes *information hiding*. Information hiding occurs when you restrict who or what has access to the data items in the group.

These groups of data items and related functions are referred to as *objects*.

Here is an example of a few objects:

```
GROUP Point

    FUNCTION  SetPositionXYZ(NUMBER, NUMBER, NUMBER)
    FUNCTION  GetPositionXYZ()

GROUP Dimension
    FUNCTION  Set-Size(NUMBER, NUMBER, NUMBER)
    FUNCTION  Get-Size()

GROUP Block

    FUNCTION  Set-Center(Point)
    FUNCTION  Set-Size(Dimension)
    FUNCTION  Get-Size()
    FUNCTION  Get-Center()
    FUNCTION  VisualInfo()
```

Notice that the data elements for these groups have been omitted. This is to illustrate a point about information hiding. Point stores a position given by 3 numbers. The position could be stored as a distance from an origin (cartesian coordinates) or as angles and a distance (polar coordinates), but with information hiding, you don't need to know these details. This implementation describes one pair of interfaces for storing and retrieving the position in cartesian coordinates. Adding direct support for polar coordinates requires just the addition of another pair of interfaces:

```
    FUNCTION  SetPositionP(NUMBER, NUMBER, NUMBER)
    FUNCTION  GetPositionP().
```

The Block could have been described by a coordinates for the center and a height and width for the dimension. But by using a Point and a Dimension, Block is not limited to a 3-dimensional cartesian coordinate system. For example, Point can be changed to use polar notation without affecting Block. Likewise, Dimension can be changed to include mass without requiring any changes to Block.

A function that is passed a Block, may obtain the (x,y,z) coordinates by asking Block for the point at its center with Get-Center(). Then it can obtain the coordinates by asking Point for its value in cartesian coordinates.

This is what is called *data abstraction*. Data abstraction shifts the focus from:

"What am I doing with this data?"

to

"What is the data that I am manipulating?"

Another useful tool of OOP is *inheritance*. Inheritance allows the programmer to create a variation of an existing object without writing it from scratch. To create a Block that has a color, this is all that is needed:

```
GROUP ColorBlock
    INHERITBlock
    FUNCTION SetColor(Color)        FUNCTION GetColor()                    FUNCTION
VisualInfo()
```

ColorBlock will use the same dimension and positioning routines as Block. From ColorBlock, the programmer might create still other types of Blocks:

```
GROUP MetalBlock
    INHERITColorBlock
    FUNCTION SetMetal(Metal)        FUNCTION GetMetal()                    FUNCTION
VisualInfo()
GROUP WoodBlock
    INHERITColorBlock
    FUNCTION SetGrain(Grain)        FUNCTION GetGrain()                    FUNCTION
VisualInfo()
```

If all of these groups were to describe objects for a rendering system, it could become very cumbersome for the rendering program to need to know about every form of a Block. For this, OOP uses what is called *polymorphism.*

As far as the rendering system is concerned, there is only a Block. And a Block has a function called VisualInfo() which returns some data to the rendering system about the appearance of the Block. The children of Block redefine VisualInfo() to return appropriate descriptions.

Polymorphism is the ability to reference an object whose type may only be determined at runtime. The rendering system can treat all of the various children of block as being type block and trust that the correct VisualInfo() will be called.

Benefits of Object Orientation or, "What's in it for me?"

To establish how OOP can help the software engineer, it's best to begin by establishing some software engineering goals or concerns and seeing how OO can help realize these goals.

- **Correct/verifiable**
  Software needs to be correct and it is desirable to have the ability to prove that it is correct. Attempting to prove the correctness of a systemis impossible if you cannot prove the correctness of each of its parts.OO will not automatically provide you with a provably correct system. It does however provide a means of addressing correctness at the component level through the use of assertions. Assertions generally come in three flavors: preconditions, postconditions, and invariants. A precondition is an initial requirement on entry to a routine (e.g., "intuition.library" must be open before calling OpenScreen()). Postconditions provide a check that the routine has achieved its goal (e.g., did OpenScreen() return a non-NULL pointer?). Finally, invariants provide a set of boundaries that the object agrees not to violate. For example, a routine that manipulates an 16-color screen may have an invariant that requires the number of a bitplane to fall in the range of 0 to 3.

- **Compatible (with similar products)**
  The more popular of today's OOPLs generally use C as an intermediate language. While in a pure OOPL, some might consider it tantamount to heresy to link in procedural elements, in practice C and assembler are still often used for performance reasons. Over time, as the compilation systems continue to develop and execution environments become faster and more complex, this will become less necessary and may eventually cease altogether. (In particular, C++ and Objective-C, being closely related to C may eliminate the need for "pure" C code. And, it is conceivable that CPUs will one day reach a level of sophistication in instruction and data pipelining that it will be beyond the capability of a human being to write assembly code that rivals the compiler generated code.)

- **Efficient**
  The ability to link with impure languages such as C and assembler provides a short term means of writing highly efficient code. In time, as compilers and machines advance, this will become a non-issue.

- **Portable**
  By using data abstraction, it is possible to isolate the machine, hardware, or operating system dependencies to a few objects. These objects may serve as ancestors to new objects that will implement the interface on a new platform.

- **Be easy to use**
  There is a great deal of ongoing research into ways of simplifying the development process. Some languages include sophisticated class browsing tools that allow the developer to observe the interactions between classes and examine the interfaces defined.
- **Maintenance is a large percentage of software engineering life cycle**
  Some estimates place the percentage as high as 70%. As explained above in the discussion of reusability and extensibility, OO allows testing to be more focused and in theory will reduce testing time and development time. Additionally, polymorphism easily permits the extension of an application in ways that were not conceived of during the original development.
- **Robust**
  (with respect to abnormal cases)
- **Extendible**
  (with respect to changes/modifications, important for large-scale programs)
- **Reusable**
- **Have integrity**

OO has the ability to address the last four issues through encapsulation, information hiding, inheritance and polymorphism. Consider the difference in adding support for a new account type to first, procedural code, then to OO code.

Procedural

```
if accounttype == Savings
    SavingsProcessing();
...
```

Add this every place it switches based on the account type:

```
else if accounttype == MoneyMarket
    MoneyMarketProcessing();
```

OOPL

```
OBJECT Account
    FUNCTION deposit();

OBJECT Savings
    INHERIT Account
```

Simply derive a new object that redefines the deposit routine. No other changes are required...

```
OBJECT MoneyMarket
    INHERIT Account
    REDEFINE   deposit
                    ... require initial deposit to be > $1000
```

because the code handling deposit transactions calls the accounts deposit function, which in turn resolves to the redefined deposit function when the account is a MoneyMarket account.

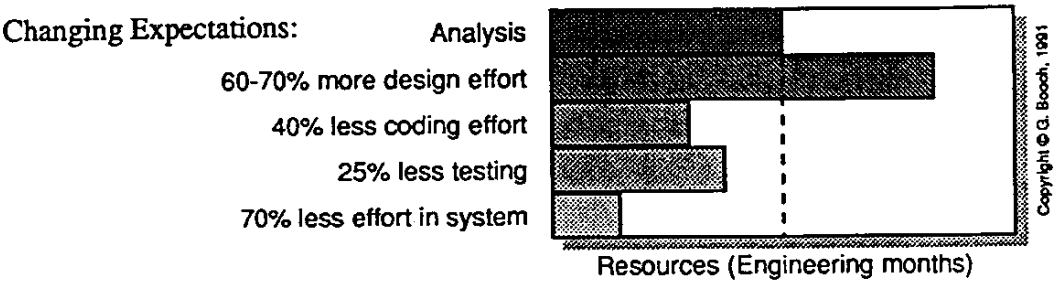
```
account.deposit();
```

The interface has been extended to support a new object without altering any of the code in the existing interface. By limiting the scope of code changes in this manner, the risk of adversely affecting the system as a whole is minimized. Since the new functionality does not alter the existing interfaces, if the new account is proven to comply with its assertions, the resulting system may be assumed to be proven. In terms of actual development and testing, OOP has removed the need to exhaustively test the existing functionality.

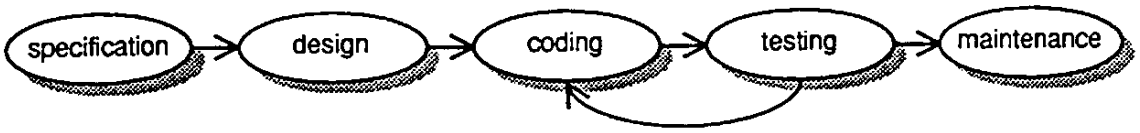
> *Beware*, there are certain assumptions in operation! The account object and the object that has contracted for its services must be in agreement on the interface, including the handling of error conditions (which may occur if an attempt is made to make an initial deposit of less than $1000 to a money market account).

OO Development

A word to the newcomers... If you are a C programmer who is interested in learning C++ (and to a lesser degree this is true for Objective-C) it is suggested you begin by learning Smalltalk or Eiffel first. By using a language that is not closely related to C you prevent yourself from slipping back to non-OO programming practices. If this is not an option, taking a good course in OOP with an emphasis in the language you have selected is strongly recommended.

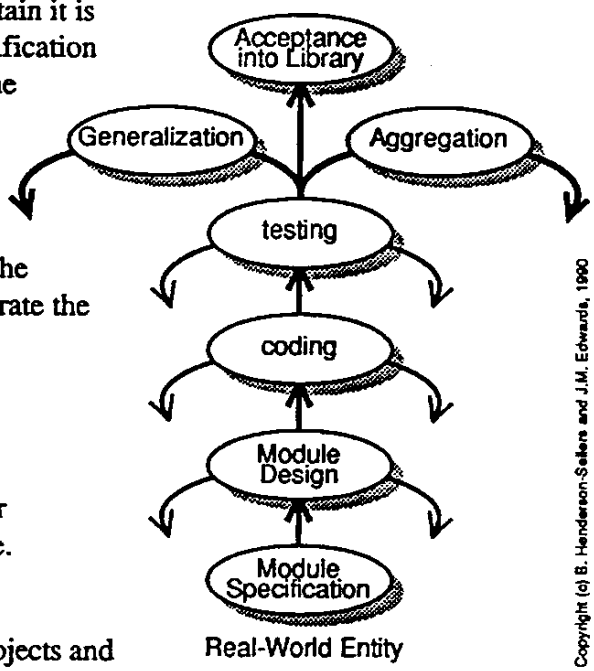


Traditional view of Development Life Cycle



OO Development Life Cycle "Fountain"

Take special note that at any point in the fountain it is possible to return all the way back to the specification phase! There is no implicit assumption that the specification is correct as there often is in the traditional, procedural design process. This is not to say that the traditional approach is wrong for procedural programming. Rather, it is typically much easier to enumerate what the desired actions of a system than it is to enumerate the objects involved and their interrelationships.



Phases of OOA

- ❑ **Specify systems requirements**
  Search the customer's requirements for "things" and the services they provide.

- ❑ **Identify the objects/classes**
  Which identified "things" constitute objects and which constitute attributes of objects?

- ❑ **Identify the relationships**
  Generalization: this is a _____.
  Association: this uses _____.
  Aggregation: this has a _____.

- ❑ **Design the relationships**
  Define what services each object offers and what services it requires.

- ❑ **Look in existing library for appropriate objects**
  Does a class exist that represents this object or could serve as a parent for this class?

- ❑ **Look for inheritance relationships**
  What elements are in common between the objects? Do these elements form the basis of a common parent object?

- ❑ **Generalize**
  Can the new objects be generalized? What concepts offer potential for reusability?

Repeat until the class(es) are acceptable for use as a library

Step 1

Systems requirements specification (SRS)

- in language of users
- source documents to find objects

Step 2

Find candidate objects
(real-world objects)

- First pass -- nouns in SRS
  - + verb → methods
  - + adjective → attributes
- Concrete *and* abstract
- (Coad & Yourdon, 1990; Nerson, 1990)
  - e.g., Structure and classification
    - Events
    - Roles played
    - Location
    - Organization

Step 3:

Establish Interactions

Analysis relationships: Classification
Association, Aggregation

Step 4:

Analysis merges to Design

Design relationships: Client-server
(with current OOPLs)

Step 5

Explore Existing Library Classes

Refine Design -- now highly detailed

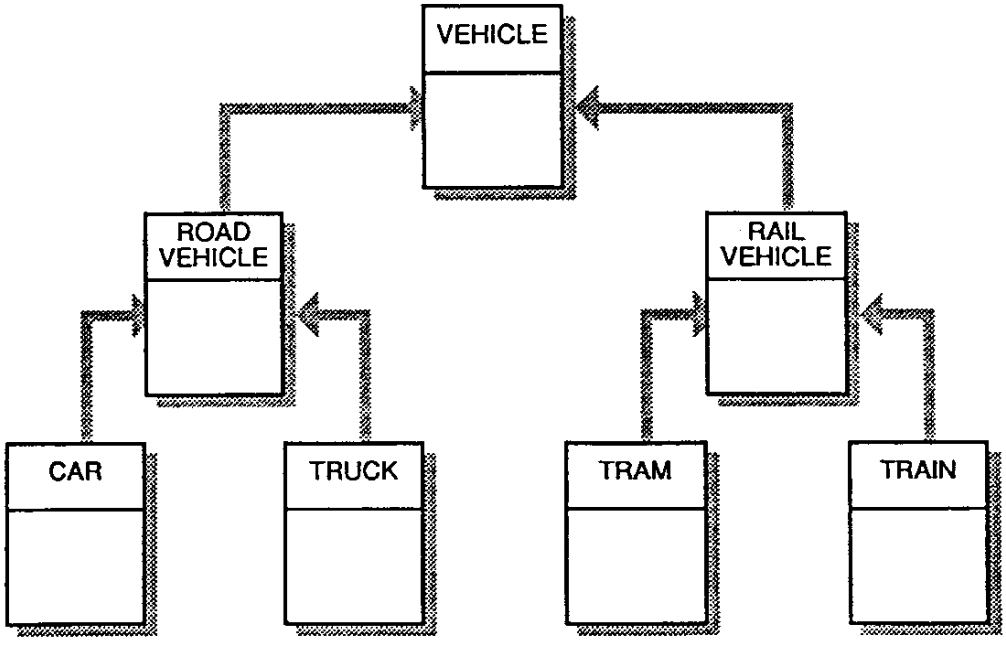
Step 6

Examine Class Network for more
Inheritance Structures

This may introduce new classes and
new interactions

Class Coded and Tested

OFF-LINE INHERITANCE HIERARCHY



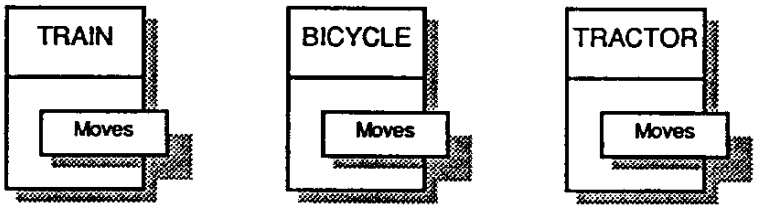
Step 7

Further Refinement beyond demands of current project in order to facilitate later reuse

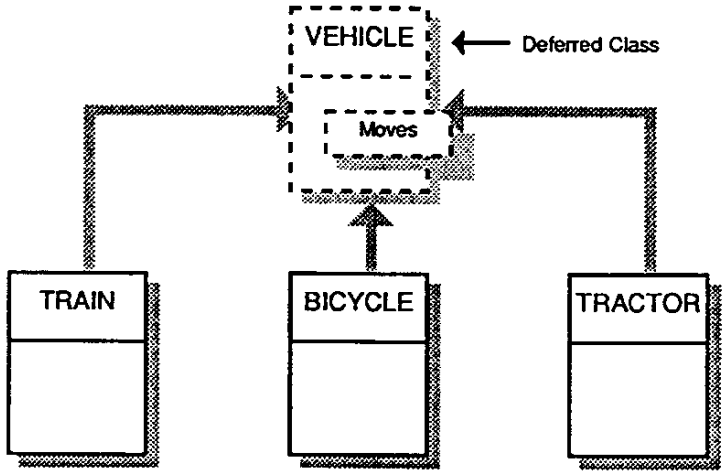
Cluster Identification

Documentation

WE FIND WE HAVE DESIGNED THE CLASSES:



GENERALIZE TO



Methodology

Step 1. Requirements Specification

e.g., customers deposit or withdraw cash or checks into one or more accounts. Available accounts are:

a) Checking Account
b) Passbook Account
c) Savings Investment Account
d) Security Plus Investment Account
e) Term Deposit Account
f) Short-Term Call Account

Step 2. Identify Entities

Fairly easy:

1) Customer
2) Checking Account (A)
3) Passbook Account (B)
4) Savings Investment Account (C)
5) Security Plus Investment Account (D)
6) Term Deposit Account (E)
7) Short-Term Call Account (F)

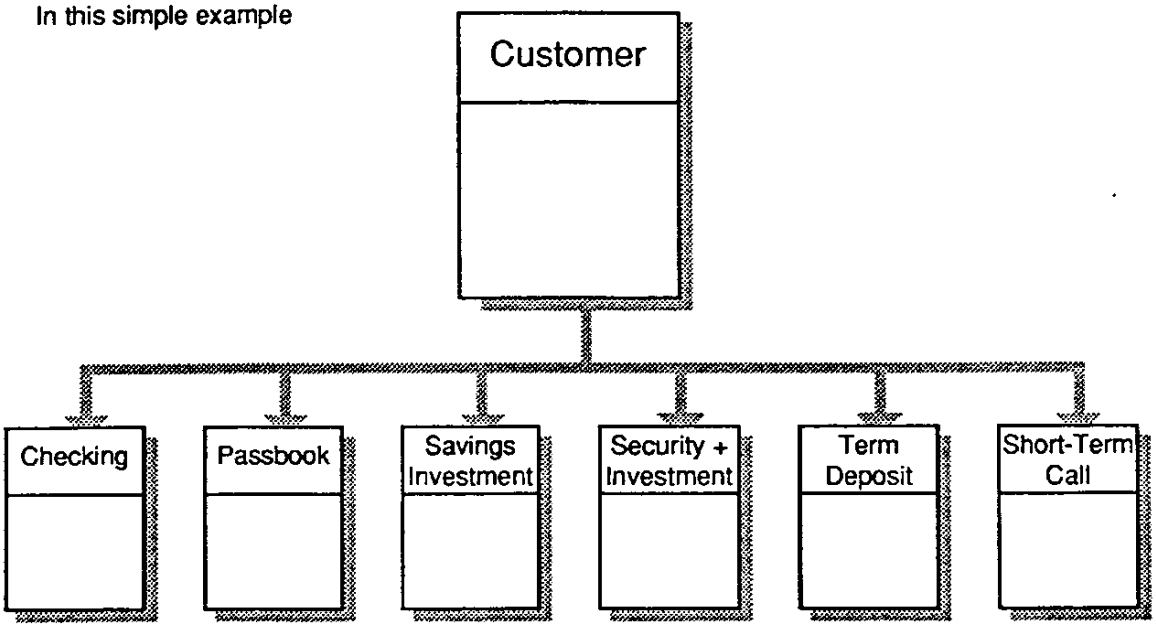
But what about:

8) Cash
9) Check

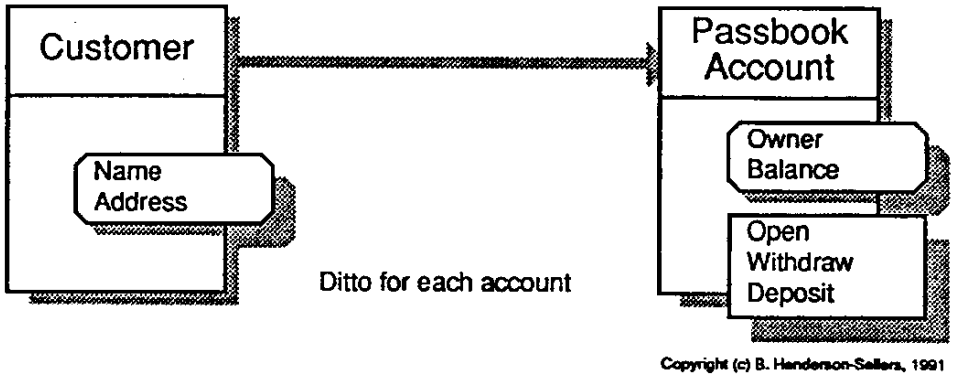
These could be attributes or objects in their own right

Step 3. Establish Interactions

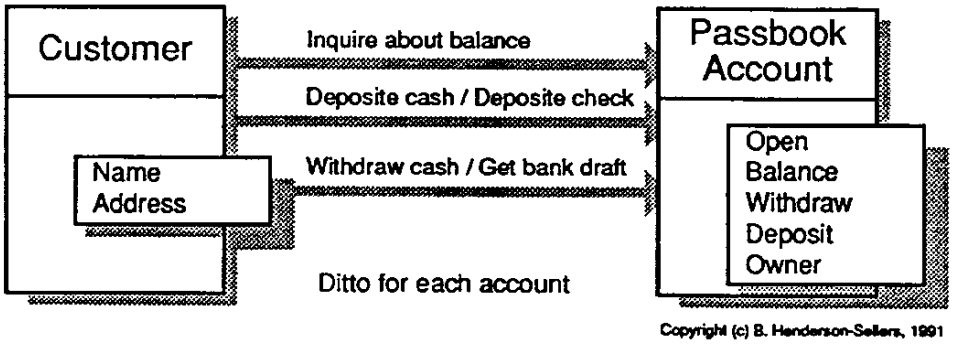
In this simple example



Add Attributes and Operations



Step 4. More Detailed Design



Step 5. (Not really here, since starting from scratch)

We might look for library classes to represent

a) case as a REAL

b) account_name as a STRING

c) possibly address as several fields (an "address" class)

Step 6. Look for Inheritance Hierarchies

Required knowledge

a) No interest on Checking Accounts. Must maintain minimum $250 balance.
b) Passbook. No minimum balance. Interest rate dependent on balance.
c) Savings Investment. Minimum balance $500. Withdrawals must be $100 or more. Interest rate dependent on balance.
d) Security Plus Investment. Minimum balance $5,000. Minimum transaction $500. Interest rate dependent on balance.
e) Term Deposits. Fixed term. Minimum deposit $500. Interest rate depends on both term and deposit.
f) Short-Term Call Account. Minimum deposit $10,000. Minimum period of deposit 7 days. All transactions minimum $1,000. Interest calculated daily (single rate).

Table of Interest Rates

Passbook

$1 - $1,999	10%
$10,000 - $19,999	13%
$2,000 - $9,999	12%
>= $20,000	15%

Savings Investment

$500 - $1,999	9.5%
$2,000 - $9,999	12%
$10,000 - $19,999	13%
>= $20,000	14%

Security Plus Investment

$5,000 - $9,999	12%
$10,000 - $19,999	13%
>= $20,000	15%

Term Deposits

Term	$500-$49,999	>= $50,000
1 mth to <3 mths	15%	15.5%
3 mths to <6 mths	16%	16%
6 mths to <12 mths	16.5%	16.5%
12 mths to <13 mths	17%	17%
13 mths to <33 mths	18%	18%
33 mths to <60 mths	18%	18%

Short-Term Call Account

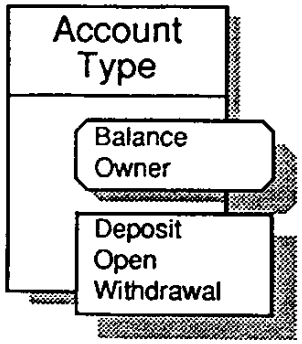
11%

One Suggested Route

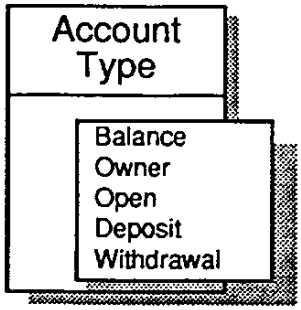
Construct an object class for each type of account

"Hidden" is how deposits are made, how withdrawals are made, how balance is calculated, current interest rate structure, etc.

ANALYSIS ICON



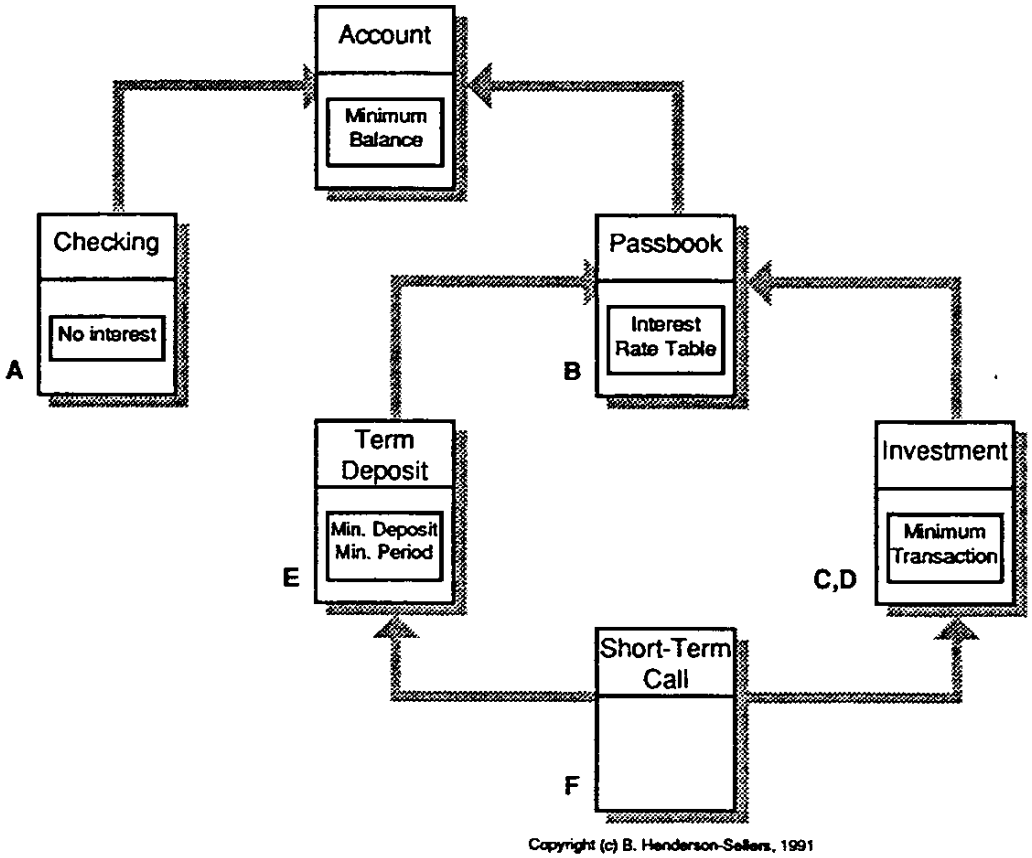
DESIGN ICON



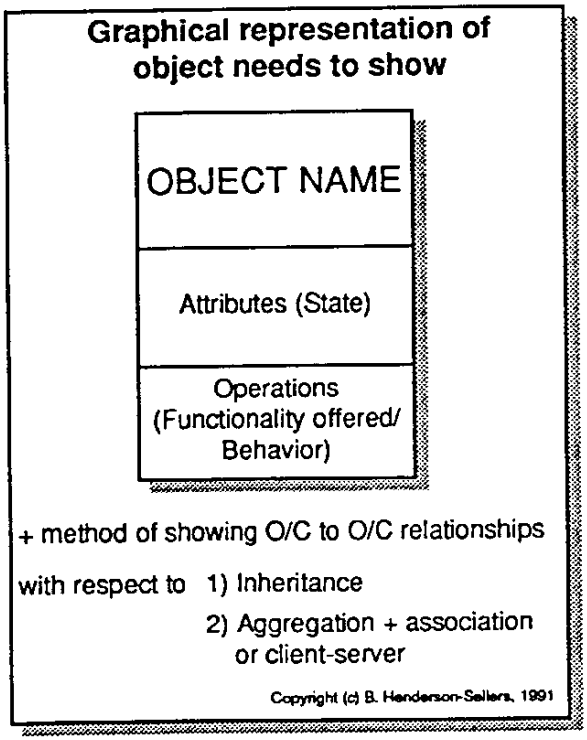
Recognize commonality

		Interest	Flat Rate Interest	Interest Table	Value of Min. Transaction	Value of Min. Transaction	Min. Deposit	Min. Period of Deposit
A	Checking				250			
B	Passbook	✔		✔				
C	Savings Investment	✔		✔	500	100		
D	Security Plus Investment	✔		✔	5,000	500		
E	Term Deposits	✔		✔	(500)		500	✔
F	Short-Term Call Account	✔	✔		(10,000)	1,000	10,000	✔

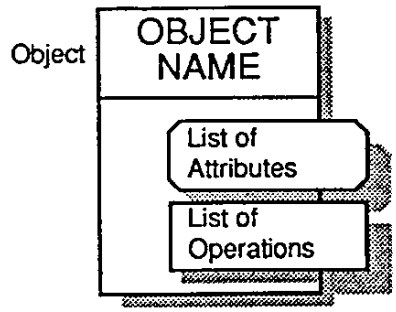
Generalization

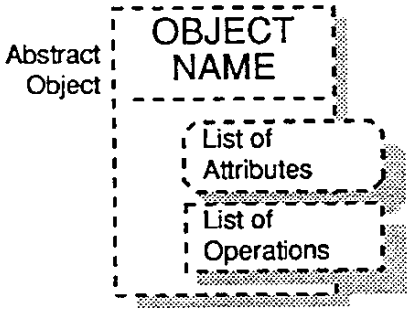


A Useful Notation for OO Analysis and Design



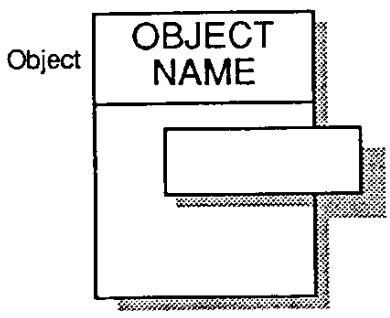
ICONS FOR ANALYSIS

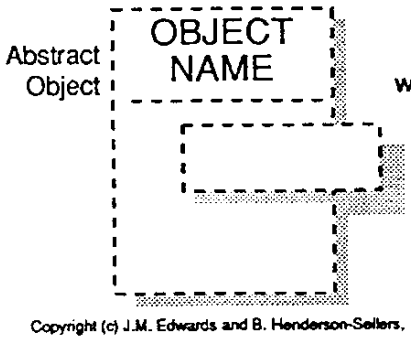


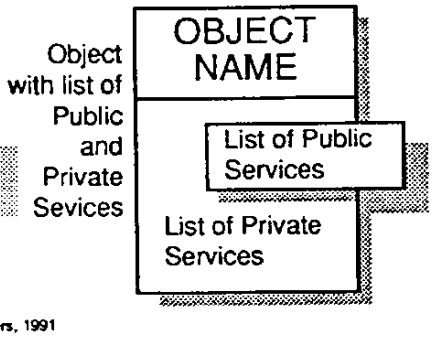


Copyright (c) J.M. Edwards and B. Henderson-Sellers, 1991

ICONS FOR DESIGN

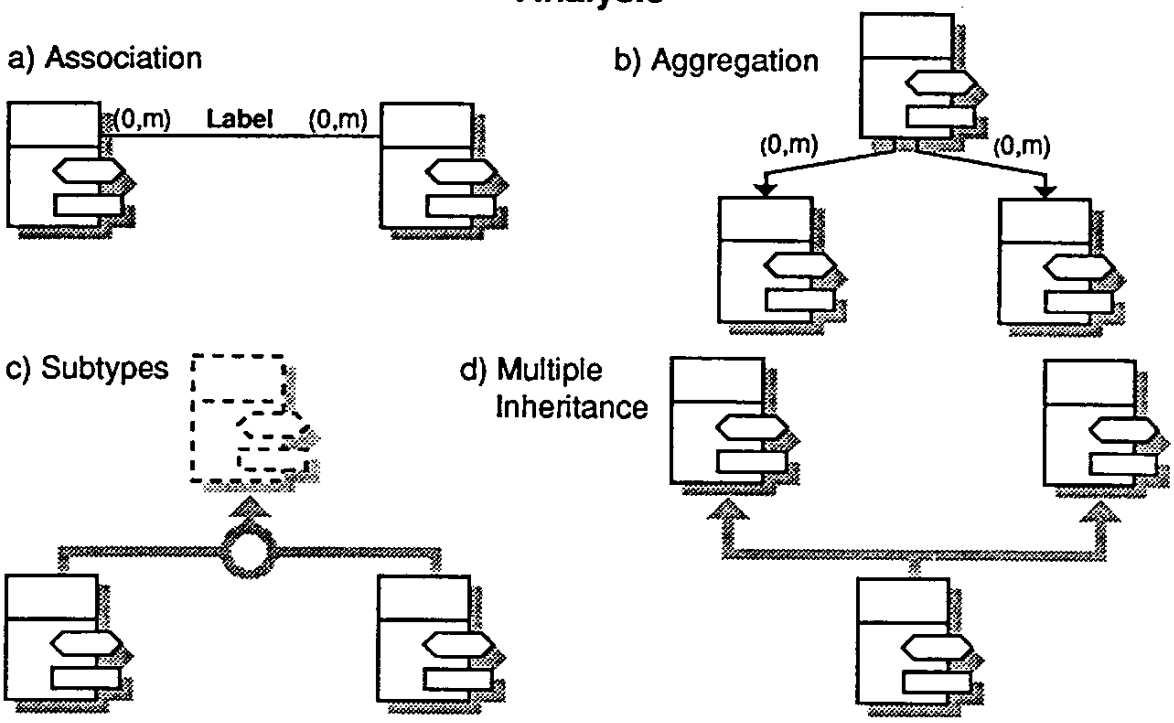




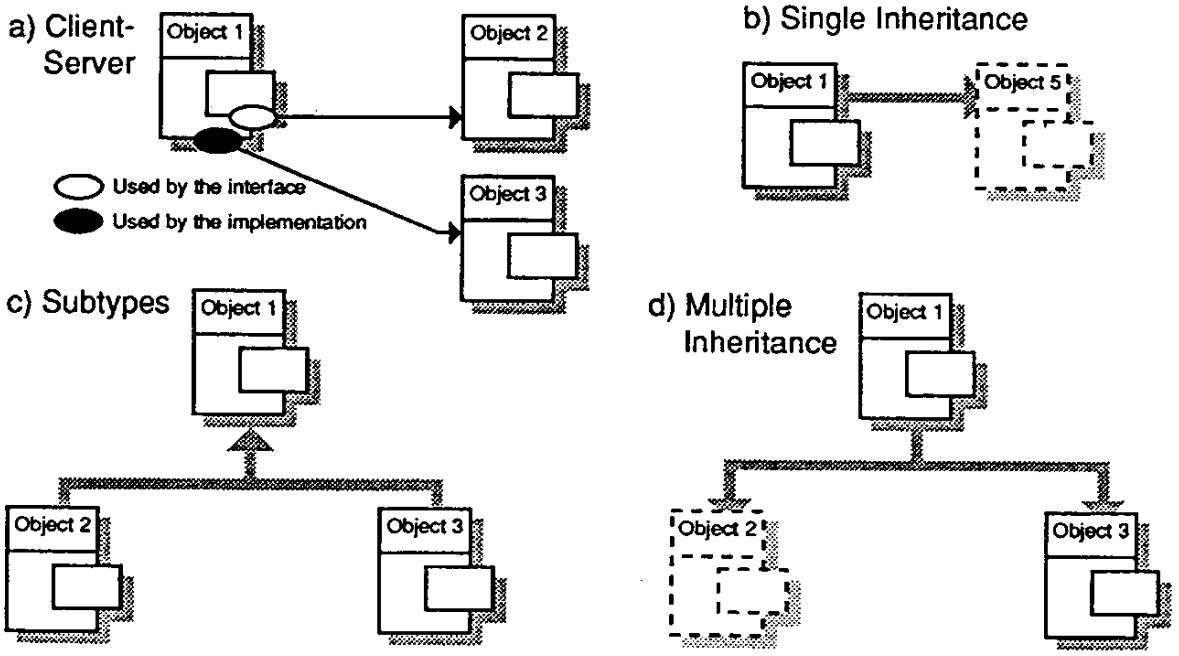


Copyright (c) J.M. Edwards and B. Henderson-Sellers, 1991

Analysis

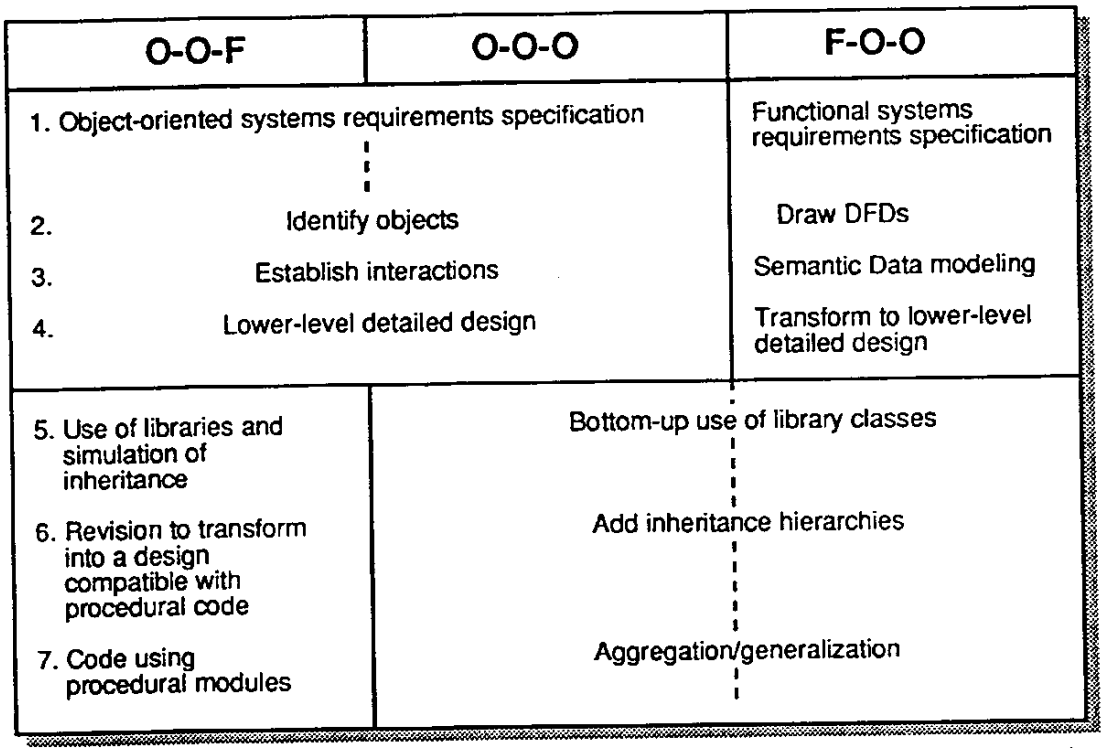


Design



Alternative approaches to Analysis and Design

While a purely OO approach is desirable, it is not always possible. It is however possible to develop using OO techniques from a non-OO specification. Likewise, it is possible to carry out the full OOA/OOD process but then implement the software using a functional language such as C.



O-O-F refers to OO analysis and design followed by a functional programming implementation.
O-O-O is the pure Object-Oriented approach to software development.
F-O-O describes OO design and implementation from a functional specification.

Another useful technique is the use of CRC cards.

CRC = Class, Responsibility, Collaboration

A technique for the earliest stage of the design process, CRC cards provide a very simple mechanism for exploring relationships between objects and identifying key objects, collaborations, and systems.

This is primarily intended as a group exercise in the style of brainstorming. CRC cards are a useful exploratory tool, but are not really suitable for the later design work.

Summary

Object-oriented programming is not the answer to all the woes of the software world. And simply switching to an object oriented programming language will not magically make your code more maintainable. Object-orientation is not a programming language or a different way of analyzing a problem. OO is a philosophy. It requires a new mindset, new approaches, new timelines. Design and analysis become the focus instead of the implementation. So take the time to do it right, learn the new ways of thinking. The potential long-term savings are significant.

◆

Product Testing

by John Kominetz

Testing is essential for creating a successful product in today's computer-savvy market. Users do not respond kindly to visits by the Guru anymore, and professionals won't tolerate losing a day's work to a software error. Testing builds quality software ; quality inspires consumer confidence and trust; consumer trust leads to sales and financial gain for the developer.

In a tight economy, harsh deadlines and staff reductions often weaken the testing process, which is rarely seen as integral to the development process. Hopefully the following will help improve the internal testing process and provide an inexpensive alternative to complement it.

Beta Testing

Harrison's *Improving the Software Testing Process* defines testing as "a systematic exercise of system components to find and classify errors with a minimum of time and effort." Testing should be an orderly, well-planned process much like a scientific experiment. Its only goal is to find and classify errors. Testing should reduce development time and maintenance releases by getting the problems out early.

Most traditional texts on testing say the same thing; I find this definition is too narrow, however. Beta testing--usually "seat-of-the-pants" testing--contributes many error reports without rigorous procedures; varied expertise, systems environments, and tester enthusiasm compensate for the formless nature of it. Beta testing is not a substitute for rigorous system testing. Rather, it complements and enhances it.

Testing is more than finding and classifying errors. It is often the conscience of the development process, reminding those under pressure to produce that quality cannot be overlooked for the sake of a deadline. Quality and reliability are the strong foundation on which to build a good product concept. Without this foundation, the greatest product concept will not sell if it crashes or loses data with alarming frequency.

Beta testing is any testing outside the development test process. It often involves personnel not employed by the developer. Beta testers can employ whatever testing methods they wish--from regimented test-case planning to just playing around; they rarely submit test plans or test logs, and their only real requirement is to submit error reports.

Advantages of Beta Testing

Relatively speaking, beta testers are free. The biggest expenses are finding the beta testers, distributing updates to them, collecting reports from them, and compensating them for their efforts. Beta testing is more a hobby than a job; do not expect to draw a salary as a beta tester, nor spend 40 hours a week beta testing.

Beta Testers can range from inexperienced users to professionals using the product in their businesses. Product assurance or testing departments tend to employ technical people, and simple errors can slip by such departments because of assumptions new users don't have, like trying to load a GIF file into a spreadsheet program. Aside from experience, beta testing allows access to many hardware and software system variants that the developer may not have. Users have old motherboards, strange CPU add-ons, outdated operating systems, emulators, odd things in their startup-sequences, etc. Amigas in particular are blessed with multi-tasking without memory protection, making it easy for interaction between applications. Further, consider how many things can be stuck into an A500 expansion slot or even the memory expansion slot!

Some would argue that many of these products--odd accelerators or unusual programs--cause the many errors reported by beta testers, not the developer's own project. When in doubt, the user blames whatever application has the error requester on it. A few lines in the manual about known incompatibilities may save users from conflicts ahead of time, or at least let them know who is really at fault.

Disadvantages of Beta Testing

Their are two major disadvantages to beta testing: First, as more people not involved with the development process obtain the product, the likelihood of piracy increases. Sometimes companies will issue unique serial numbers as part of their product/beta test package so they can track leaks, but this is time consuming. Second, without direct control over the tester's procedures, it is impossible to know the quality of testing being done. Ultimately, only the number of legitimate error reports shows how effective a beta tester is. Be prepared for some testers that submit no error reports because they lost interest. This second reason is why regular testing is critical to the development process; without it there is no way to get a feel for how well a product has been tested.

Choosing Beta Testers

The defining characteristic of a good beta tester is interest in the product so he will spend valuable free time testing it. Experience and exotic equipment can be useful, but the tester

must spend time testing the product to find errors. There are four broad classes of beta testers, each with advantages and disadvantages, but they all require the basic enthusiasm or need to test the product to be good testers. The four types are as follows:

Company Employees

If the company is large enough, it is possible to get people not associated with the development process to use the pre-release product. Most companies already have non-disclosure agreements with their employees, eliminating some of the legal mumbo jumbo required to use non-employees. Upgrading a version of the beta software can be as easy as leaving a disk on someone's chair. Getting or clarifying error reports is similarly easy when the person is accessible for 40 hours per week.

Users

The bulk of beta testers come from common every-day users, preferably the kinds of users that would buy the product. Even with non-disclosure agreements, this is how many beta versions of a product find their way to pirate bulletin boards. Upgrading and collecting reports can be difficult with this class of tester. Worse, the user probably has a job, maybe even a spouse or family, reducing the amount of time available for testing the product. Still, through friends and user groups, these are the most accessible candidates for beta testers.

Other Developers

Other developers in a similar but non-competitive market can provide more sophisticated testing than the average user. If the beta product is designed to cooperate with another developer's product, behind-the-scenes information can uncover errors quickly. The only problem with using other developers is that they are usually as manpower-short or deadline-weary as you are! Approaching individuals in other companies can give you a user-style beta tester with exceptional technical knowledge. Be careful that beta testing your product does not constitute a conflict of interest for the tester!

Professional Users

This is the most productive, and potentially the most annoying, candidate for beta testing. Seek out companies that could use your product as part of their business. Nobody will stress a product as much as somebody using it 40 hours a week. Here, the tester's interest is replaced by something far more demanding--the user's needs. Unlike normal beta testers, this class of tester will be eager to report errors, and even more eager to receive newer versions with bug fixes. Be prepared for many phone calls. Also be prepared to provide rapid technical support, workarounds, and quick-fix versions. While this can be the most effective kind of beta tester, expect him to demand more support and interaction from you.

Getting Started

To weed out beta test candidates, require them to fax or write an official request to become a beta tester. Make sure the letter states that upon receiving the initial beta test packet, he agrees to be bound by a non-disclosure agreement. Marginally interested candidates will not be bothered with such "formalities." These are often the people who cannot be bothered with submitting reproducible error reports or other such "formalities."

Once the letter is received, send the tester a complete beta test package including the product, current documentation, release notes, a non-disclosure agreement--to be signed and returned immediately, and information on error reporting. The information on error reporting should include a standardized error report form, guidelines for writing a reproducible error report, and information on who to contact with error reports and questions. Preparing a brief guide to testing and reporting errors can save much time and effort, especially if it eliminates the need to clarify ambiguous reports.

Depending on how often new versions are available, a beta tester should get a complete test package, sans non-disclosure agreement, with each new beta release. This does not mean that you should send testers every new version of the product; designate special landmark builds as new beta test material--maybe three or four during the entire development process.

Compensating Beta Testers

It is not unusual to offer extra incentives to beta testers. The most common form is to give the tester a copy of the finished product. Having a prize for the individual with the most reproducible errors can build a healthy feeling of competition among the testers, especially if the top three scores are distributed with the latest test package. Often, having early access and direct technical support for a new product is enough to satisfy most beta testers. Use whatever feels most comfortable.

More important, keep the names of effective beta testers for future product testing, and share this information with other developers. Develop a good database of testers, and you will have a valuable resource without investing all the start-up costs each time.

Conclusion

Beta testing can be an inexpensive and effective form of alternative testing. Careful preparation and research on your part are required, but the result is a higher quality product. Beta testers can become much more than just testers, even as a source for new employees or business alliances. Remember that beta testing itself is not enough, and that rigorous product testing as part of the development process is still required.

◆

Using Amiga Debugging Tools

by Carolyn Scheppner & Adam Levin-Delson

Have you ever experienced any of the following problems with software ?

> The software runs well on your system but some others report it has problems on their systems.
> The software runs well by itself but has problems running with or after other software.
> The software runs well most of the time but occasionally crashes or fails for no apparent reason.

You can find and eliminate many of these types of hidden software problems by running Amiga debugging and stressing tools while developing and testing your product.

Hidden and obvious software problems are often caused by use of null pointers, uninitialized pointers, improperly initialized structures, improper use of I/O requests, improper abort code, improper memory usage, or overwriting of memory allocations. By using Enforcer, Mungwall, IO_Torture, Scratch, Memoration, and other Amiga debugging and testing tools, you can catch most of these problems before you ship a product.

In fact, many companies now require that all of their in-house software pass at least Enforcer and Mungwall testing, and have also added this requirement to their contracts for outside development. Many other CATS tools such as Devmon, Owner, and LVO can help to pinpoint the causes of problems. You have these tools. Here are hints on how to use them, when to use them, and how to get the most from their output.

Enforcer and Mungwall

Enforcer and Mungwall are the top two bug finding and bug prevention tools. Enforcer is an MMU-based debugging tool by Michael Sinz, based on an earlier tool by Bryce Nesbitt.

An MMU is a memory management unit which can be configured to trap accesses to specified ranges of memory. The 68030 and 68040 processors have a built-in MMU, and most 68020 boards contain a separate MMU. Because it is MMU-based, Enforcer can trap reads and writes of low memory and non-existent memory the instant these accesses (also known as "Enforcer hits") occur. This allows you to catch usage of null pointers and some

uninitialized pointers in your program, and even accesses which would have trashed low memory or otherwise crashed the system.

Some of these accesses may seem harmless on your system (such as reads of address 0) but could cause your program to fail in the field. If you are developing commercial software (or any software that you plan to distribute) it is extremely important that you invest in a MMU, or at the very least make sure that your software is tested on machines with an MMU, Enforcer, and Mungwall. As more of the development community begins running these tools, software that is unusable in their presence will simply not be used. The main differences between the new Enforcer and the previous Enforcers is that the new Enforcer also works with 68040 processors and has a wide variety of output options built in. In addition, due to the variety of output options, the new Enforcer must be *run*. The new Enforcer requires at least V37 OS, so if you need to run Enforcer on a 1.3 machine, you must use the older Enforcer 2.8b which we will probably be renaming *Enforcer1.3*. Full docs are provided with Enforcer.

Enforcer is even more powerful when used in combination with Mungwall. Mungwall is a combination memory munging tool by Ewout Walraven which is based on Memmung by Bryce Nesbitt and Memwall by Randell Jesup. The "mung" part of Mungwall fills all of free memory (and all subsequently freed memory) with a nasty odd 32-bit values like `$DEADF00D`. Such a values are almost guaranteed to cause serious problems for any program that uses uninitialized pointers or structures, or uses memory or allocations after they are freed. Such usages can occur, for example, when allocations are not freed in the correct order.

Mungwall uses a few different nasty 32-bit values in its memory munging to help you diagnose any problems.

- Except when Enforcer is running, location 0 is set to `$C0DEDBAD` so that programs referencing location zero will not find a value. Programs referencing location 0 as a string will get a string of high ASCII characters rather than a null string, and programs using null structure pointers should be irritated into crashing. When Enforcer is running, this is not necessary because with location 0 containing 0, Enforcer can trap these low memory accesses by itself.
- On startup all free memory is munged with `$ABADCAFE`. If this number shows up, someone is referencing memory in the free list.
- Except when MEMF_CLEAR is set, memory is pre-munged on allocation with `$DEADFOOD`. When this is seen in an Enforcer report, the caller is allocating memory and doesn't initialize it before using it.

- Memory is filled with $DEADBEEF when it is deallocated, encouraging programs reusing freed memory to crash or get Enforcer hits.

The "wall" part of Mungwall allocates extra memory before and after every memory allocation and fills this "wall" with a fill pattern and other information. On each de-allocation, Mungwall checks to make sure that the deallocation size matches the size of the allocation, and checks to make sure that the walls have not been overwritten. Mungwall also watches for 0-size allocations, 0-size deallocations, and 0-address deallocations. Mungwall checks for incorrect allocations (such as 0 size) during AllocMem, and checks for incorrect deallocations or trashing around allocations during FreeMem. If trashing or incorrect deallocation is detected, Mungwall will both report it and refuse to free the memory. These reports are all known as "Mungwall hits".

The latest Mungwall can also optionally report on failed allocations with the SHOWFAIL option. In addition, Mungwall has an option to "snoop" and report on all memory allocations and deallocations for all tasks or specified tasks. This can be useful when tracking down memory losses. The sometimes voluminous snoop output can be run through the snoopstrip program which will throw away all matching allocation/deallocation pairs.

Another useful recently added Mungwall feature is the NAMETAG option. When this option is active, Mungwall will tag all memory allocations with the first 16 characters of the name of the task, process, or command that allocated the memory. A program called Munglist can then do its best to show the names of the allocators of currently allocated pieces of memory. This can be very useful in tracking down memory losses. However, keep in mind that some of an application's allocations may be done by a different system task (for example, if an application opens a file, a file handle or buffers may be allocated by the process for the disk volume).

Mungwall may be used without Enforcer and on non-MMU machines. If you don't have an MMU, at least test with Mungwall alone. If you are using uninitialized memory or memory after it is freed, Mungwall should help to crash you immediately (as you might crash on a user's machine when he runs other programs at the same time as yours). Mungwall is more pleasant when used with Enforcer however, since it will generally incite an Enforcer hit when memory is misused, rather than a crash.

It is generally not safe or recommended to turn Mungwall off and back on without rebooting. This is because Mungwall recognizes its walled allocations by looking for a special value it has placed near the beginning of the allocation. If a program allocates memory while Mungwall is running, then frees it while Mungwall is turned off, and another program reallocates the some of the same memory while Mungwall is turned off, and then tries to deallocate it with Mungwall turned back on, Mungwall may generate Mungwall hits due to

incorrect data. Note that this can also occasionally happen on a soft-reboot if a task that starts up and allocates memory before Mungwall is run happens to receive a Mungwall-tagged piece of memory that was in use before reboot, and then frees the memory after Mungwall is running. To identify such problems, turn your machine off for 15 seconds or so to clear memory, turn it back on, and retest.

A useful alternative to turning Mungwall off and on is the Mungwall UPDATE feature. UPDATE allows you to turn on and off many of the options of another running copy of Mungwall. For example, you could turn off the MEGASTACK option via Mungwall UPDATE NOMEGASTACK or turn on the SHOWFAIL option with Mungwall UPDATE SHOWFAIL.

Debugging Terminals

Enforcer and Mungwall both output their debugging information to the serial port at the baud rate your machine's serial hardware is set to. At powerup, your serial hardware is set to 9600 baud, but you can modify this by bringing up a terminal package and setting a baud rate. The optimal debugging setup is to connect your Amiga via a null modem serial cable to another Amiga or computer running a terminal package with ACSII capture capability. Both Enforcer and Mungwall include Ctrl-Gs in their output to generate a beep with most terminal packages, and the ASCII capture capability will allow you to capture all serial debugging output to a file for examination. This is especially useful when combined with serial kprintf() (debug.lib) debugging statements in your code such as kprintf("About to close window $%lx",win).

But some developers don't have a second machine to use as a debugging terminal, or may need to debug a serial application which would interfere with serial debugging output. So two other options exist. One is to use the parallel versions of the debugging tools, and the parallel debugging *dprintf* command (from ddebug.lib) and connect a parallel printer for the output. The benefit of serial and parallel debugging is that you can get some information out and capture it even if your machine is crashing, and even if your application has taken over the system. The other option is *Sushi*.

Sushi

For most applications, an option now exists for debugging on a single machine. It's called *Sushi* (a late-night name loosely based on the fact that it captures "raw" serial output). Sushi captures all raw serial debugging output (such as kprintf(), new Enforcer with RAWIO option, Mungwall, IO_Torture, etc.) and has its AmigaDOS process send the debug information to stdout. This means you can redirect Sushi's output to a an AUTO/CON

window (and the window will pop open if you have any hits), or to a multiserial port, etc. Sushi can also be told to save its circular buffer (via Ctrl-F signal or via a separate SUSHI SAVEAS FILENAME command), or to empty its buffer (Ctrl-E signal or a separate SUSHI EMPTY command).

Sushi also can operate in quiet mode, just capturing debugging output, and provides a program task signalling interface which allows an external application to be written to display the captured debugging output (for those of you who want a fancier debugging display program). A benefit of Sushi is that fairly voluminous debugging output can be used even from within interrupt code or intense task code because Sushi only has to place the text bytes in a ram buffer, and the actual output is done by a separate process. See Sushi.doc for more information and a source for a sample external display program.

Sample user-startup:

```
run >NIL: Mungwall
run >NIL: Enforcer RAWIO
run >NIL: sushi <>"CON:0/20/640/100/Sushi CTRL-File CTRL-Empty/AUTO/CLOSE/WAIT" NOPROMPT
```

Debugging printf()s: kprintf() and dprintf()

When using serial (or parallel) debugging and stress-testing tools, it is very useful to have your own program's debugging statements (such as "About to do xxx") mixed in with any potential hits from the debugging tools.

In addition, it is very useful to have a printf()-like command that can be used from within even the low level task or interrupt code. A clean way to add conditional debugging statements to a C program is to use a MACRO such a D(bug)) by including lines like the following in your program. Set MYDEBUG to 1 to turn on debugging. Set bug to printf for printf debugging, or to kprintf (and link with debug.lib) for serial debugging, or to dprintf (and link with ddebug.lib) for parallel debugging. The D(bug()) macro is neater in your code because it can be indented and you need not surround it with any #ifdef directives yourself. Just be careful to remember to put two close parens at the end of each D(bug()) statement, before the semicolon.

```
/**********    debug macros     ***********/
#define MYDEBUG  1
void kprintf(UBYTE *fmt,...);
void dprintf(UBYTE *fmt,...);
#define DEBTIME 0
#define bug printf
#if MYDEBUG
#define D(x) (x); if(DEBTIME>0) Delay(DEBTIME);
```

```
#else
#define D(x) ;
#endif /* MYDEBUG */
/********** end of debug macros **********/

Example macro usage:
win = OpenWindow(&mynewwin);
D(bug("Opened window at $%lx", win));
```

Note that kprintf() and dprintf() debugging can be used inside even the lowest level task or interrupt code (although you better keep output down to a minimum during interrupts!). The DEBTIME (Delay) in the macro above must be 0 however if you are doing output from anything other than a Process.

Finding the Cause of Enforcer and Mungwall Hits

By using Enforcer and Mungwall while you are developing your software, you can catch problems as soon as they are introduced and greatly cut down your debugging time. It is especially useful to place conditional remote debugging statements in your code as you write each routine so that they can easily be turned on when a problem occurs. You will easily be able to pinpoint the problem area when the kprintf() (or dprintf()) output is intermixed with the Enforcer or Mungwall output. The remote debugging commands kprintf() and dprintf() are available in the linker libraries debug.lib (serial) and ddebug.lib (parallel) respectively. These linker libs are supplied with some compilers and are also available on Commodore's Native Developer Update disks.

Some people prefer to use a source-level or single-stepping debugger in combination with Enforcer and Mungwall when tracking down a problem to single-step through their code until the hit occurs. A different low-level method of locating Enforcer or Mungwall hits is to disassemble program memory where the hit occurred. If the hit occurred in ROM, first try using OWNER to determine ROM subsystem of that address. For example, OWNER 0x202442 might return the following on a soft-kicked A2500:

```
Address          Owner
00202442         in resident module: exec 39.47 (28.8.92)
```

Then use LVO to determine the probable function at that address within the subsystem:
`LVO exec romaddress=0x202442`

Closest to `$202442` without going over: `exec.library LVO $fe0e -498`
OpenResource() jumps to `$202358` on this system

(See the Owner and LVO section for more information on these tools)

If this does not give enough clue, try disassembling just before other ROM and RAM addresses shown in the Enforcer stack dump line. If code is found, these may be application or ROM functions which called the routine that generated the hit.

If the hit occurs in RAM code, use various methods to match up the disassembly with your own code. Assembly programmers can just compare the disassembly to their source. Others may wish to take the hex values of a sequence of position-independent 68000 instructions near the hit (i.e., no addresses except for offsets and branches) and do a search for this binary pattern in your object modules. If you find the pattern, do a mixed source and object disassembly of that object module (for example, with SAS's OMD you could OMD >ram:dump mymodule.o mymodule.c) and then look in the output for instructions matching those where the hit occurred. Note that when a hit occurs in a disk-loaded device or library, it is currently not possible to determine who owns the code where the hit occurred. This is because it is up to the device or library to store its seglist wherever it sees fit in its own library or device base. If you suspect that your hit is occuring in a disk-loaded library or device, you can try searching suspected disk libraries or devices for a match of the hex pattern of position-independent code near the hit.

Deciphering Enforcer and Mungwall Output

Enforcer and Mungwall provide lots of valuable information to help debug your hits.

Sample Enforcer Output:

Here is a sample Enforcer hit caused by a program called *lawbreaker*.

```
WORD-READ from 00000014                          PC: 075899A6
USP:  075A8BE0 SR: 0015 SW: 0769  (U0)(F)(D)  TCB: 075AE468
Data: DDDD0041 DDDD1100 DDDD2200 DDDD3300 DDDD4400 DDDD5500 DDDD6600 DDDD7700
Addr: AAAA0000 AAAA1100 AAAA2200 AAAA3300 AAAA4400 AAAA5500 07400810 --------
Stck: 00000009 00000008 00000007 00000006 00000005 00000004 00000003 00000002
Stck: 00000001 00F92A18 00001000 075AEE4C BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB
Name: "Shell Process"  CLI: "lawbreaker"  Hunk 0000 Offset 00000096

LONG-WRITE to  FEEDFACE        data=DDDD0041   PC: 075899B8
USP:  075A8BE0 SR: 0018 SW: 0709  (U0)(F)(-)  TCB: 075AE468
Data: DDDD0041 DDDD1100 DDDD2200 DDDD3300 DDDD4400 DDDD5500 DDDD6600 DDDD7700
Addr: AAAA0000 AAAA1100 AAAA2200 AAAA3300 AAAA4400 AAAA5500 07400810 --------
Stck: 00000009 00000008 00000007 00000006 00000005 00000004 00000003 00000002
Stck: 00000001 00F92A18 00001000 075AEE4C BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB
Name: "Shell Process"  CLI: "lawbreaker"  Hunk 0000 Offset 000000A8
```

Here is an explanation of the some of the important information:

PC:
Program Counter. This is the address in memory where the program was executing instructions when the hit occurred. For some kinds of hits this will often be the address of the instruction after the hit. Note that if your program passes a bad pointer or an improperly initialized structure to a system ROM routine, you may cause ROM code to read or write to an illegal address.

TYPE-SIZE from or to
(e.g., WORD-READ from 00000014):
This is the type of illegal access and the address where it occurred. In the first example, the illegal access occurred at address `$14`, and is specified as a WORD-READ access. This means the illegal memory access was an attempt to read a word (2 bytes) at address `$14`. Low memory accesses are often caused by NULL pointers to structures. If, for example, your code or a ROM routine references a structure member at offset `$14`, and what you provided or used a NULL structure pointer, Enforcer will pick up a hit at address `$14`. Other illegal READ accesses are BYTE-READ (generally caused by a bad string pointer), and LONG-READ (generally caused by a bad pointer or a bad pointer within a structure). An Enforcer hit will occur whenever a program attempts to read low memory addresses (generally caused by a NULL pointer of some type) or non-existent memory addresses (generally caused by an uninitialized pointer). When Enforcer catches an illegal READ access, it reports the access and prevents the program from reading the address by returning data of zero.

On illegal WRITE accesses (i.e., attempts to write to ROM, low memory, or non-existent memory) Enforcer will report BYTE-WRITE, WORD-WRITE, or LONG-WRITE hits (such as in the second example). These mean the access would have illegally and dangerously written to memory which should not be written to, quite possibly causing memory to be trashed. WRITE hits can be caused by bad pointers or bad code. Fortunately, Enforcer prevents the actual memory trashing from occurring, and instead just reports it. This can allow debugging severe low memory trashing problems which would normally crash your machine. In the second example above, the CLI program *lawbreaker* tried to write the long value `$DDDD0041` to the address `$FEEDFACE`.

Occasionally you will see an INSTRUCTION access of illegal memory meaning the PC (Program Counter) is at an address it has no legal reason to be at. Generally this is caused by trashed code, a trashed return address on your stack, or an invalid library base. Hint - when an INSTRUCTION Enforcer hit occurs at a negative address, it is most often caused by a

JSR to an LVO (negative word offset) from a NULL library base. Check the Stack dump lines for a possible return address to the code that made the bad library call.

"data=xxxxxxxx":
On WRITE hits, Enforcer will show you the data that would have been written to the illegal address.

Register Dump Lines:
All of the middle Enforcer lines show the contents of the program's registers and stack at the time of the hit. The Data: line shows the D registers (D0 though D7). The Addr: line shows the A registers (A0 though A6). A7 (the Stack register) is shown as USP, and the contents of the top of the stack is shown in the Stck: lines (note - more Stck: lines are available via a command line option of the new Enforcer). The USP line also contains the contains of additional processor status registers, and also the address of the Task Control Block (TCB). On the same line are symbols which specify whether a Forbid (F) and/or a Disable (D) was in effect when the hit occurred.

Name and Hunk Offset:
Example: Name: "Shell Process" CLI: "lawbreaker" Hunk 0000 Offset 000000A8

This line shows the task name, and CLI command name (if any) of the task or command that was executing when the hit occurred. If possible, Enforcer also provides a hunk offset to where the program counter was if the hit occurred within the program's own code instructions.

Sample Mungwall Output

Here are sample Mungwall hits by a program called *mungwalltest*. Additional explanation has been added in parentheses below each hit. For reference, the arguments for memory functions are AllocMem(size,type) and FreeMem(address,size). The A: and C: addresses are Mungwall's guess at the address from which AllocMem was called. The A: address is the address if the caller was assembler code. The C: address is the probable address if the caller was C code, assuming a standard stub. Since Mungwall is wedged into the memory allocation functions, it can only guess the caller's address based on what is pushed on the stack. The "task" address on the first line of a Mungwall hit is the task address of the caller. Mungwall checks for incorrect allocations (such as 0 size) during AllocMem, and checks for incorrect deallocations or trashing around allocations during FreeMem. If trashing or incorrect deallocation is detected, Mungwall will both report it and refuse to free the memory.

Note that Mungwall has special code to prevent trapping the partial (wrong size) deallocations that are performed by some version of layers.library. If any other debugging tools are also wedged into AllocMem and FreeMem, Mungwall's A: and C: addresses may be thrown off by additional information pushed on the stack, and Mungwall will also be unable to screen out any partial layers deallocations (which will show up as hits on your task's context).

```
AllocMem(0x0,10000) attempted by 'mungwalltest' (task 0x3E08F0)
from A:0x3EBB12 C:0x3EF552 SP:0x4559FC
```
(means mungwalltest tried to allocate 0 bytes of memory)

```
FreeMem(0x0,16) attempted by 'mungwalltest' (task 0x3E08F0)
from A:0x3EBB40 C:0x3EF598 SP:0x4559F4
```
(mungwalltest tried to free memory it never got - i.e., at address 0)

```
FreeMem(0x2FE7C0,0) attempted by 'mungwalltest' (task 0x3E08F0)
from A:0x3EBB40 C:0x3EF5A8 SP:0x4559EC
```
(mungwalltest tried to free 0 bytes of memory)

```
Mis-aligned FreeMem(0x2FE7C4,16) attempted by 'mungwalltest' (task 0x3E08F0)
from A:0x3EBB40 C:0x3EF5B6 SP:0x4559E4
```
(mungwalltest tried to free memory at an incorrect address)

```
Mismatched FreeMem size 14!
Original allocation: 16 bytes from A:0x3EBB12 C:0x3EF574 Task 0x3E08F0
Testing with original size.
```
(mungwalltest tried to free a different size from what it allocated)

```
19 byte(s) before allocation at 0x2FE7C0, size 16 were hit!
>$: BBBBBBBB BBBBBBBB BB536572 6765616E 74277320 50657070 65722000
```
(memory before this allocation was trashed; trash shown at right)

```
8 byte(s) after allocation at 0x2FE7C0, size 16 were hit!
>$: 75622042 616E6400 BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB
```
(memory after this allocation was trashed; trash shown at left)

```
FreeMem(0x2FE7C0,14) attempted by 'mungwalltest' (task 0x3E08F0)
from A:0x3EBB40 C:0x3EF5F4 SP:0x4559D0
```
(deallocation refused due to trashing explained in last two reports)

```
Failed memory allocation!
AllocMem(0x500000000,1) attempted by 'flush' (task 0x3E08F0)
from A:0x453E96 C:0x1 SP:0x4820F4
```
(SHOWFAIL option shows flush tried to allocate memory and failed)

As you can see, Mungwall alone can catch a large variety of memory-related software problems. But one of the most important benefits of Mungwall is that by filling freed memory with nasty 32-bit values, it can force subtle memory misuse problems into the open, by often causing the misuses to access addresses that can be trapped on by Enforcer.

Enforcer and Mungwall are required tools for the development of bug-free Amiga software. If you don't have an MMU, get one. The investment in an A3000, 68030 card, or 68020+MMU card will quickly pay for itself by cutting down on your development and allowing you to catch and find software problems with Enforcer. Enforcer and Mungwall are not just for developers and QA departments. Anyone who uses or reviews in-house software or software for purchase or contract can benefit your company by catching hidden software problems during normal usage and examination of the programs. Many people at Commodore and other companies run Enforcer and Mungwall all of the time. Keep that in mind if you are trying to impress a company with your software!

Other Debugging and Testing Tools

The next three tools, Memoration and Scratch by Bill Hawes, and IO_Torture by Bryce Nesbitt are important QA tools for testing the robustness of all application failure code, for uncovering illegal register usage in assembler code, and for detecting unsafe use of device I/O requests. Following notes on these three are notes on some of the additional debugging and stresstest tools provided by CATS.

Memoration

Memoration by Bill Hawes is a tool to selectively limit the ability of a task or module to allocate memory, thereby simulating the effects of a low-memory condition. It provides the unique ability to selectively cause each individual memory allocation of a program, whether direct or indirect, to fail, thereby allowing you to test the failure path and abort code at every point in your application code.

Memoration works by SetFunctioning the Exec AllocMem() and/or AllocVec() entries and then screening the requests. If a request from a particular task or range of addresses is received, memoration returns a zero instead of passing it through to AllocMem().

When a task or module is denied a memory request, memoration sends a message to the serial port identifying the client task ID, the address it was called from, and the size of the denied request. If the software can't handle being denied its memory request, this message will typically be followed by a series of enforcer reports telling of how the software went ahead and wrote to location 0.

Command-Line Parameters.

Memoration accepts command-line parameters to specify the module or task name and the range of memory sizes to disallow. The argument template is

```
MODULE,TASK/K,CLI/K/N,OFF/S,MIN/K/N,MAX/K/N,AFTER/K/N,
EVERY/K/N,ALLADDR/S,ALLOCVEC/S,CHIP/S,FAST/S,BACKTRACE/K/N
```

and the specifications can be changed at any time by reissuing the command.

MODULE is the name of a ROMTag or library.
The resident modules are searched first, followed by a search of the system library list. When an entry is found, the range of addresses encompassing its code is determined using several methods. For ROMTags the range extends from the ROMTag itself to the next higher module, or to RT_ENDSKIP if no higher module exists. For libraries a certain amount of voodoo is required, as the location of the library ROMtag isn't stored in the (public) library structure. In this case memoration examines the LVOs to determine the lowest and highest addresses, and then searches for a ROMtag in the range (low-$2000,high+$2000). If a ROMTag is found, memoration uses the smaller of the ROMTag address and the lowest LVO address as the low limit, and the larger of the RT_ENDSKIP address and the highest LVO address as the high limit.

TASK specifies the name of a task to trap.
The task must exist at the time memoration is run, and for best results should persist for the course of testing. If you're using WShell (as you should be) you can define a name for a particular shell instance by using "newwsh name sucker".

CLI specifies a shell number as the task to trap.

MIN specifies the minimum memory request to trap.
The default is 0.

MAX specifies the maximum allocation to trap.
The default is 2000000.

OFF turns off memory trapping.
The code patch is left intact, but won't trap any requests until enabled again. AllocMem() and AllocVec() traps can be turned on and off separately.

ALLOCVEC sets the trap for the AllocVec() entry, instead of AllocMem(). Both functions can be trapped independently.

AFTER specifies the number of allocations (within other specifications) to pass before beginning the trap.

EVERY traps every Nth allocation meeting the specifications.

ALLADDR sets the address range to all memory.

CHIP limits the trap to Chip memory specifications.

FAST limits the trap to Fast memory specifications.

BACKTRACE specifies the number of longwords of stack backtrace desired.

Examples:

```
memoration myprog after 15  ; after 15th allocation, deny myprog any memory
memoration dos.library  ; disable all DOS allocations
memoration task DF0 min 400 ; disable larger allocations by DF0:
memoration icon.library task Workbench every 3
memoration console.device min 40 backtrace 8
```

Example test session to test failure of every allocation an application makes:

Open two shells - one for memoration (denoted by 1> below) and one for myprog (denoted by 2> below). Alternately, you could start myprog from Workbench. Run Enforcer and Mungwall. Have serial or Sushi debugging set up.

1> memoration myprog after 0
2> myprog ; a failure occurs but hopefully no hits or crashes
1> memoration off (exit myprog if it made it up)

1> memoration myprog after 1
2> myprog ; a failure occurs but hopefully no hits or crashes
1> memoration off (exit myprog if it made it up)

1> memoration myprog after 2
2> myprog ; a failure occurs but hopefully no hits or crashes
1> memoration off (exit myprog if it made it up)

Note - until you get up to a higher "after" number, myprog will likely fail during the startup code it is linked with - i.e., before even reaching your main() entry point.

Further Notes. Memoration uses a seglist-split for its code patch, and so shouldn't be made resident, at least not on the first execution. Memoration was written by William S. Hawes.

Scratch

Scratch by Bill Hawes purposely trashes the scratch registers (D1, A0, A1) on exit from library functions of any "scratched" library. This will point out assembler callers that accidentally or purposely depend upon getting "useful" values from those registers. It is extremely important that ALL assembler applications be tested with Scratch. Even minor changes to the OS can change the values that happen to be left over in registers on exit from a system library function. Assembler code that is accidentally reusing a scratch register (for example A1) after a system call, or code that is looking at the wrong register for a result has the potential to break badly with even a minor change to the OS. Scratch can catch these problems before you ship. To use Scratch, see the script *SCRATCHALL.SCRIPT* which properly excepts certain system functions from scratching.

IO_Torture

IO_torture is a tool which can be used to check for improper use or reuse of device I/O requests.

IO_torture should be part of the standard test suite for all products.

Exec device I/O usage is tracked by IO_torture. If an IORequest is reused while still active, IO_torture will print a warning message on the serial port (parallel for IO_torture.par).

The current plan of IO_torture includes:

SendIO()

Check that message is free. Check for ReplyPort. Be sure request is not linked into a list.

BeginIO()
Check that message is free. Check for ReplyPort. Be sure request is not linked into a list.

OpenDevice()
Mark message as free. If error, trash IO_DEVICE, IO_UNIT and LN_TYPE. Be sure request is not linked into a list.

CloseDevice()
Check that message is free. Trash IO_DEVICE, IO_UNIT.

IO_torture does not currently check for another common mistake: After virtually all uses of AbortIO(IORequest), there should be a call to WaitIO(IORequest). AbortIO() asks the device to finish the I/O as soon as possible; this may or may not happen instantly. AbortIO() does not wait for or remove the replied message.

Note regarding NT_MESSAGE: NT_MESSAGE would seem to be the correct node type for an I/O request which is newly initialized. However, part of how IO_torture works is that it makes sure in-use requests are marked as NT_MESSAGE, and would normally complain if such an NT_MESSAGE came through BeginIO() or SendIO() (as it would signify reuse before ReplyMsg). So that IO_torture will not complain about a freshly initialized MT_MESSAGE I/O request, IO_torture also checks to see if the message has ever been linked into a list. If it has not, IO_torture will let the NT_MESSAAHE marked request by without complaining. This is necessary because the amiga.lib CreateExtIO() and CreateStdIO() historically set a newly created I/O request node type to NT_MESSAGE.

Devmon

Devmon is a tool which allows you to monitor the activity of any exec device. It is extremely useful for debugging both application use of a device and your own device drivers. Devmon captures (or outputs to serial or Sushi) debugging information every time any vector of a monitored device is entered. Optionally, Devmon can also report on the entry to (and in some cases, exit from) the exec library functions which call the device.

Usage:
```
devmon name.device unitnum [remote] [hex] [allunits] [full]
```

remote
means serial debugging output

hex
means show values in hex

allunits
monitor regardless of unit pointer (required if device gives a new unit pointer to every opener)

full
means also monitor exec device-related library functions

Some sample Devmon output as generated by:

```
devmon clipboard.device 0 allunits hex full remote
```

In the following regular devmon output lines, UPPERCASE: signifies entry to an actual device vector, while MixedCase: signifies entry to (or Rts from) an exec function. The number preceded by "@" is the address of the I/O request. The single letters following are abbreviations for the fields of an IOStdRequest (C means io_Command, F means io_Flags, E means io_Error, etc.). When Devmon is exited, it outputs a key to these fields.

Here we see C:ConClip executing DoIO() of a request containing CMD_WRITE (C = 3, as defined in exec/io.h) of 8 bytes (L=$8) of data at $30BDF0. We see the same I/O request make it to the BEGINIO vector of the device, and then we see the DoIO() completing successfully with an io_Actual (A=) of 8.

```
DoIO : @$478294 C= 3 F=$81 E=0 A=$4 L=$8 D=$30BDF0 O=$0 (C:ConClip)
BEGIN: @$478294 C= 3 F=$1 E=0 A=$4 L=$8 D=$30BDF0 O=$0 (C:ConClip)
DoRts: @$478294 C= 3 F=$81 E=0 A=$8 L=$8 D=$30BDF0 O=$8 (C:ConClip)
```

TNT

TNT installs a trap handler that replaces Software Error requesters with a large requester containing informative debugging information. TNT can be called in your user-startup (it does not need to be run). But it does not get along well with trap-based single-stepping debuggers. So if you plan to run a debugger, turn off TNT with TNT OFF.

TNT requesters contain PC, task/command name, SP, SSP, register, and stack contents information similar to that displayed by Enforcer.

Owner and LVO

Owner and LVO are very useful for determining the owner of a memory address. This can help you determine the location and even the cause of an Enforcer or Mungwall hit.

LVO requires the Amiga FD files for your OS version in a directory assigned the logical name FD:. The FD files are provided with the includes and linker libs for the OS.

When you get a hit, use OWNER to determine if the address is in a ROM module (such as exec or intuition), or in the loaded code, stack, port, or other memory owned by a program. Note that owner can not determine if an address is in the code of a disk-loaded device or library because there is no standard place for devices and libraries to store their seglist.

Examples:

```
1> owner 0x202443

Address          Owner
00202442         in resident module: exec 39.47 (28.8.92)
```

Now use LVO to determine the probable function at that address within the subsystem:

```
1> LVO exec romaddress=0x202442
```

Closest to `$202442` without going over: `exec.library  LVO $fe0e -498`
OpenResource() jumps to `$202358` on this system

LVO can also be used as a programming helper to list one function and its FD comment:

```
1> LVO exec AllocMem exec.library  LVO $ff3a -198  AllocMem()
AllocMem(byteSize,requirements)(d0/d1)
```

LVO will list all of a library's LVOs if no function name is provided. With the optional CONTAINS keyword, LVO will also list the addresses each function jumps to on YOUR system (different on different systems). Note that if you have debugging tools installed which use SetFunction (for example, Mungwall, Devmon, or Scratch), LVO will not be able to determine the ROM address of the SetFunctioned library functions because the LVOs of those functions will point to the SetFunctioned RAM code.

LVO can also create command lines for the *Wedge* program.

Note that LVO and Wedge can only interact with actual library functions, not for their various stub interfaces which only exist in linker libraries. Actual library functions are the functions listed in the FD file for the libraries. Note also that under new versions of the OS, some older library functions become unused by the OS and newer functions are used in their place.

Wedge

Wedge is a tool for wedging into almost any system library function and monitoring calls to and results from the function, for any or all system tasks. Sometimes it can be more efficient to quickly wedge and monitor a system function rather than ad debugging code and recompile. It used to be very difficult to create command line arguments for wedge. But LVO can generate a template command line to "wedge" into any system function.

Example: Creates a wedge command for wedging into OpenScreen

```
1> LVO intuition CloseScreen wedgeline run wedge intuition 0xffbe 0x8100
0x8100 opt r "c=CloseScreen(screen)(a0)"
```

If you execute the WEDGE command line above, you will receive serial reports on every call to CloseScreen. To turn off the wedge, type WEDGE KILLALL. See Wedge.doc for more information.

Tstat

Tstat is a handy little tool for checking the signals, priority, state, and other Task control block variables of any task or command. Tstat also does its best to show the task registers as they were saved at task-switch time, and the stack usage of the task or command. Note however that Tstat is looking at private exec Task context information and therefore often needs to be updated for each OS release, and may misinterpret the task context data in unusual conditions such as a task switch which occurs in the middle of an FPU instruction. But it is very useful when checking for lost signals, bad Forbid or Disable counts, and hung Waits. Tstat can monitor once, or periodically. For example, `tstat MyProg -4` would monitor MyProg every 4/50ths of a second until Ctrl-C is hit. In a pinch, Tstat can be used as a poor man's logic state analyzer to track another program stuck in a loop.

Other Memory Tools

EatMem

is an interactive tool for scaling back the apparent amount of memory available to the system. It is useful for testing applications in a simulated low-memory situation, as well as for testing how an application deals with only chip memory or only fast memory. EatMem requires at least V37 OS.

MemList
displays all memory blocks (both free and in-use) in the system. It can be useful for debugging fragmentation/deallocation problems. See also the NAMETAG option of Mungwall, and Munglist.

Frags
displays a chart of current system memory fragmentation.

Memmon
saves current free memory and a comment to a file on demand. It is useful for documenting memory usage while testing various program operations.

Flush
does two large allocations designed to fail and thereby cause all disk-loaded devices, libraries, and fonts which are not currently in use to be flushed from the system. Useful when doing memory-loss testing and device or library development.

Snoop
can be used to snoop memory allocations, but has largely been replaced by the more flexible Mungwall SNOOP option. SnoopStrip is used to discard allocation/deallocation pairs from captured Snoop or Mungwall SNOOP output.

Report

All bug reports, compatibility problem reports, and enhancement requests must be generated with the latest Amiga *Report* program (currently version 39.6), or must be compatible with the output of the Amiga Report program. This makes automatic submission and routing of bug reports possible. The Report program is included on most developer tool and DevCon disks. Report HELP will output instructions and submissions addresses.

> We **strongly request** that you submit your reports electronically if possible. **Please use report.** Only mail them on paper if you have no other method available.

European ADSP users
Post in appropriate closed adsp bugs topic

BIX/CIX
Post bugs in the appropriate bugs topic of your closed conference.

UUCP
to uunet!cbmvax!bugs OR rutgers!cbmvax!bugs OR bugs@commodore.COM
(enhancement requests to cbmvax!suggestions instead of cbmvax!bugs)

Mail
Mail individual bug reports in Report format, on disk if possible.

> **European developers:**
> Mail bug reports to your support manager unless your support manager says to mail directly to West Chester.
>
> **U.S./others mail to:**
> ATTN: BUG REPORTS
> Amiga Software Engineering,
> CBM
> 1200 Wilson Drive
> West Chester, PA., 19380, USA

Please make sure the initial one-line bug description you provide in each of your reports is as explicit as possible. Engineering's bug summary reports and bug processing tools often list only the one-line description for each bug. "Bug in intuition" is a useless title for an intuition subsystem bug. "Pixel trash when window dragged left" is a much better title.

The latest Report program always includes a list of the currently acceptable subsystems for bug reports and enhancement requests. These allow reports to be properly routed. The current list is:

A2024	A2065	A2090.A2090A	A2091.A590
A2232	A2300	A2410	A2620
A2630	A3000	AmigaBasic	aa.chips
alink	amiga.lib	amigaguide	amigaterm
amigavision	appshell	arexx	art
as225	asl.library	audio.device	autoconfig
battclock	battmem	bootmenu	bridgeboard
bru	bullet	cdos.command	cdtv
cia.resource	clipboard	commodities	con-handler
console.device	creditcard	crossdos	custom.chips
datatypes	debug.lib	disk.resource	diskfont
documentation	dos.library	exec	expansion
filesysres	filesystem	fonts	fountain
gadget.classes	gadtools	gameport	genlock
graphics	hardware	hdbackup	hdtoolbox
icon.library	iconedit	ide.device	iffparse
input.device	installer	intuition	iprefs
keyboard	keymap	keymaps	kickmenu
layers	locale.library	mathffp	mathieee
mathieeedoub	mathieeesing	microemacs	monitors
multiview	narrator.device	new.look	parallel.device
port-handler	potgo.resource	preferences	printer.device
printer.driver	queue-handler	ram-handler	ramdrive.device
ramlib	resource	sana2	script
scsi.device	serial.device	setpatch	shell
speak-handler	startup	strap	system.command
toolkit	toolmaker	tools	trackdisk
translations	translator	unix	util.command
utility.library	wack	wbtag	workbench

◆

Writing OS-Friendly Games

by Chris Green & Spencer Shanson

Many Amiga programmers believe that the only way to write a whiz-bang, speed-of-light game is to bypass the operating system and go straight to the metal. A better approach is to write games that are OS-friendly. The combination of OS 3.0 and the AA chipset makes this more possible than ever.

Reasons to use the OS for games

- ❑ Why reinvent the wheel? Spend your time doing things that only you can do.
- ❑ Compatibility with future chipsets. For instance, AAA is *not* register-level compatible with AA.
- ❑ Easier adaptation to future hardware. For instance, it takes less time to convert a 16-color ECS game that uses the OS into a 256 color AGA game than it does to convert a hardware-banging ECS game.
- ❑ RTG compatibility possible for some games.
- ❑ The OS automatically supports pre-ECS, ECS, and AA (and will support AAA).
- ❑ Easier integration with other system components (CD-ROM, networks, serial ports, etc.).
- ❑ Easy hard-disk install.
- ❑ Less code to write. The OS has routines for handling all screen positions and scrolls, mouse movement, etc. This means less development time and less time getting the hardware to work, and more time making the game more playable.
- ❑ More robustness. For instance, the OS floppy disk code is far less picky about drive parameters than 99% of custom floppy I/O code.
- ❑ Hides bugs and quirks of the chipset. The AA chip set has a few bugs which the OS hides from you.
- ❑ The code runs out of ROM, which is faster than running the code out of Chip RAM.
- ❑ Multiple platforms. OS code will run on all Amiga-based machines, whatever their flavor.
- ❑ Tools exist to help you debug your code rely on the OS being around (e.g., Mungwall and Enforcer).
- ❑ A properly written game can be promoted, and thus work on cheap VGA monitors.

Things the OS can't currently do

- ❑ Scroll individual scanlines of a ViewPort
- ❑ AA color copperlist fades
- ❑ Dynamically update user copper lists.

All these are planned to be addressed in future OS releases. One of our goals is to make it possible to perform as many Amiga tricks in normal Intuition screens as is possible.

ECS-AA incompatibilities that the OS handles

- ❑ Vertical counter behaves differently in programmable beam modes.
- ❑ No SuperHires color scrambling.
- ❑ Bitplane alignment problems.

AA-AAA incompatibilities that the OS will handle correctly

- ❑ AAA has no fetch-mode selections. All selections are automatic.
- ❑ Different DDFSTART/STOP calculations for single/dual systems.
- ❑ Color loading is different
- ❑ Exact copper timings are different
- ❑ No SuperHires
- ❑ Multiple blitters

Game programming problems and solutions

Q: What is the graphics rendering routines are much slower than my own blitter code?
A: Use the blitter yourself. Call OwnBlitter, do setup, call WaitBlit(), poke the blitter registers, and then DisownBlitter() when all blits are done.

> *Note*: OwnBlitter() is only 3 instructions (counting RTS) when no one else is trying to use the blitter.

Q: What if input.device eats too many cycles?
A: Install a high priority input handler which chokes off all events. This handler is also a convenient way to get keys and mouse events yourself. Simply store the raw keypresses and mouse moves in your own variables.

Q: How do I change both bitmap pointers and colors in sync?
A: Use a user-copper list to cause a copper interrupt on line 0 of yourViewPort. The copper interrupt handler will signal a high-priority task that calls ChangeVPBitMap (or ScrollVPort) and LoadRGB32 to cause the changes. This allows perfect 60hz animation on an A1200, even while moving the mouse as fast as possible, and inserting floppy disks.

Under 3.0, you can also do this in an exclusive screen. You can tell if it was your screen which caused the copper interrupt by checking the flag VP_HIDE in your ViewPort->Modes.

Q: I need to use the blitter in an interrupt driven manner instead ofpolling it for completion. Aren't the QBlit routines too slow?
A: The QBlit/QBSBlit system was completely rewritten for 3.0, and has quite low overhead.

Q: How do I determine elapsed time in my game?
A: A simple, low overhead way to determine elapsed time is to call ReadEClock. This returns a 64 bit timer value which counts E Clocks, and returns how many EClocks happen per second. If you use these results properly, you can insure that your game runs at the proper speed regardless of CPU type, chip speed, or PAL/NTSC clocking.

A1200 speed issues

The A1200 has a fairly large number of wait-states when accessing Chip RAM. ROM is zero wait-states. Due to the slow RAM speed, it may be better to use calculations for some things that you might have used tables for on the A500. Add-on RAM will probably be faster than Chip RAM, so it is worth segmenting your game so that parts of it can go into Fast RAM if available.

For good performance, it is critical that you code your important loops to execute entirely from the on-chip 256-byte cache. A straight line loop 258 bytes long will execute far slower than a 254 byte one. The '020 is a 32 bit chip. Longword accesses will be twice as fast when they are aligned on a longword boundary. Aligning the entry points of routines on 32-bit boundaries can help, also. You should also make sure that the stack is always longword aligned. Write accesses to Chip RAM incur wait-states. However, other processor instructions can execute while results are being written to memory:

```
move.l  d0,(a0)+    ; store x coordinate
move.l  d1,(a0)+    ; store y coordinate
add.l   d2,d0          ; x+=deltax
add.l   d3,d1          ; y+=deltay
```

will be slower than:

```
move.l  d0,(a0)+ ; store x coordinate
add.l   d2,d0          ; x+=deltax
move.l  d1,(a0)+    ; store y coordinate
add.l   d3,d1          ; y+=deltay
```

The 68020 adds a number of enhancements to the 68000 architecture, including new addressing modes and instructions. Some of these are unconditional speedups, while others only sometimes help.

Addressing modes

- **Scaled Indexing**

  The 68000 addressing mode (disp,An,Dn) can have a scale factor of 2, 4, or 8 applied to the data register on the 68020. This is totally free in terms of instruction length and execution time. For example:

```
68000                          68020
add.w  d0,d0                   move.w (0,a1,d0.w*2),d1
move.w (0,a1,d0.w),d1
```

- **16 bit offsets on An+Rn modes**

  The 68000 only supported 8 bit displacements when using the sum of an address register and another register as a memory address. The 68020 supports 16 bit displacements. This costs one extra cycle when the instruction is not in cache, but is free if the instruction is in cache. 32 bit displacements can also be used, but they cost 4 additional clock cycles.

- **Data registers can be used as addresses**

  (d0) is 3 cycles slower than (a0), and it only takes 2 cycles to move a data register to an address register, but this can help in situations where there is no free address register.

- **Memory indirect addressing**

  These instructions can help in some circumstances when there are not any free register to load a pointer into. Otherwise, they lose.

New instructions

- **Extended precision divide an multiply instructions**

  The 68020 can perform 32x32->32, 32x32->64 multiplication and 32/32 and 64/32 division. These are significantly faster than the multi-precision operations which are required on the 68000.

- **EXTB. Sign extend byte to longword**

  Faster than the equivalent EXT.W EXT.L sequence on the 68000.

- **CMPI and TST**

  Compare immediate and TST work in program-counter relative mode on the 68020.

- **Shift instructions**

  On the 020, all shift instructions execute in the same amount of time, regardless of how many bits are shifted. Note that ASL and ASR are slower than LSL and LSR. The break-even point on ADD Dn,Dn versus LSL is at two shifts.

- **Bit field instructions**
  BFINS inserts a bitfield, and is faster than 2 MOVEs plus and AND and an OR. This instruction can be used nicely in fill routines or text plotting. BFEXTU/BFEXTS can extract and optionally sign-extend a bitfield on an arbitrary boundary. BFFFO can find the highest order bit set in a field. BFSET, BFCHG, and BFCLR can set, complement, or clear up to 32 bits at arbitrary boundaries.

Hardware resources

- **Blitter**
  Use OwnBlitter()/DisownBlitter() to claim and relinquish ownership of the blitter.

You must use the graphics.library WaitBlit(). This is as fast as possible, uses no CPU registers, and knows about blitter bugs. You cannot possibly write one that is more efficient and works on all Amigas.

- **Copper**
  If you really have to take over the copper, get the LoadView(NULL), do two WaitTOF()s, and then install your own copperlists in the cop1/2jmp registers. We do not recommend this, though. Future chipsets may have faster and more efficient coppers (AAA has a 32 bit copper!), and we will want to use these. If you load the old copper registers behind graphics' back, we have no way of switching back to the old 16-bit mode.

```
temp=GfxBase->ActiView;
LoadView(NULL);
WaitTOF();
WaitTOF(); /* custom.copllc = ??? */
...
WaitTOF();
WaitTOF();
LoadView(temp);
custom.copllc=GfxBase->copinit;
```

- **Audio**
  Use the Audio device. There are functions to change the volume, period, frequency, data etc., that is played on any of the channels. If you must hit the audio hardware, you can ask for the channel you need with the highest priority (127), and the audio channel will never be stolen from you until you give the channel back to the system.

- **Timers**

  Use the timer device. Some of the timer.device functions work as libraries, and so are easy to use. This allows you to be compatible should we use, say, a third CIA timer.

The vertical blank can be used as a special low-frequency timer. See below. CIA timers can be allocated via the resource allocation calls. The "Resources" chapter of the *2.0 ROM Kernel Reference Manual: Devices* has a good example. of CIA timer allocation.

- **Input**

  Input will usually come from keyboard, mouse, joystick, infra-red etc. Mouse and joystick can be easily read from the hardware keyboard input could come from the keyboard.device, which knows how to handle keyboard timings, but it is easier by far to open an intuition window and read either RAWKEY or VANILLAKEY IDCMP messages. These either give the raw key number pressed, or the character the key pressed represents (useful for international games).
- **Interrupts**

  Set up interrupt servers with high priority. Your server will then be the first called.
- **Disk drives**

  Just use the dos.library. It's so much easier, works on all possible drives, past, present and future, and makes software so much more friendly to the user. Floppy based copy protection can be done by allocating the blitter and inhibiting the drive while checking for the key track.

Do's and Don'ts

- *Do* clear unused bits when writing, and mask out unused or unneeded bits when reading.
- *Don't* use timing loops. The reasons should be obvious.
- *Don't* write self-modifying code unless you know how instruction caches work.
- *Don't* steal memory. You can always call CloseWorkbench().
- If you are hardware banging, don't assume anything about the initial contents of the display registers when your program is started. Initialize everything.
- If using ViewPorts, be sure to have a properly allocated ViewPortExtra. Some graphics calls are faster when one is present.
- *Do* note well the warning around the copinit structure.

CPU Differences

- ❑ Caches.
- ❑ Copyback and write-through modes.
- ❑ Access to Chip RAM.
- ❑ '020, '030, '040 instruction and effective addresses.
- ❑ MMUs and FPUs

Conclusion

Writing OS-friendly games benefits you, the developer, and the gamers who buy your games. It ensures the viability of your code on all Amiga platforms, not just the one you used for development. By allowing the OS to do the mundane details of opening screens and windows, you give yourself more time to do the creative portions of your game. It does not matter how fast your game is if the gameplay is lousy and the artwork looks like a finger painting from a pre-schooler.

An OS-friendly game is no longer something to be avoided. We've provided ample tools in the OS to support your gaming needs, and we will continue to do so. The sooner you write OS-friendly games, the sooner you'll be prepared for the future.

◆

CDTV Programming and Spooly Device

by Dave Parkinson

Outline

When CDTV was launched in 1990, most of the development community came rapidly to the conclusion "it's just an Amiga". About six months later, with the experience of the first generation titles behind us, we came to the conclusion "Oh no it isn't". It became clear that despite being based on a logical extension of standard Amiga hardware and operating software, the CDTV posed some unique problems, and that effort would have to be put into creating tools to help solve these problems. This document discusses work done in the UK in this area in the last few years, with special reference to a tool called "spooly device". A lot of this work has been based around two conferences held in Buxton in the Derbyshire Dales, and has been assisted by a UK (now European) multimedia research group known as "the Buxton group".

The Problems of CDTV Development

Experience with early CDTV title development indicated three particular problem areas. These were as follows:

- **Development environment.**
  Every experienced CDTV programmer knows stories of teams who developed CDTV products using high-level authoring tools on things like Amiga 3000s, then eventually cut a test gold disk near the end of the project, and were absolutely horrified by the results. It's clear that you need early testing on the CDTV itself. But gold disks are expensive (around £400 each in England up to very recently), and that's a lot of money to spend finding out you made a trivial mistake in your startup-sequence!

- **The remote controller**
  The CDTV IR controller has been responsible for a new recommended style of user-interaction known as the "bouncing highlight", but this is not straightforward to implement. In order to access things like the HELP key and the B button it is necessary to operate the controller in MOUSE mode

where the arrow keys generate a stream of mouse-moves, which isn't at all convenient when it comes to implementing a bouncing highlight. The situation got worse with the introduction of trackballs, IR Mice and A570 drives - it turns out that contrary to original expectation a high proportion of a titles' users will be using real mice after all, in which case the bouncing highlight isn't particularly appropriate and may even be unusable!

- **Long user wait-states during seeks**
  The biggest factor leading to horror on viewing early test-disks (and even some released disks) was an unacceptable rate of user-response due to long seek-times on CD. A program which needs to access a lot of small files representing buttons or brushes before it continues may perform fine on harddisk, but totally unacceptably on CD - user "wait states" of over forty-five seconds have been reported in some cases. It's quite clear this is unacceptable to put it mildly!

Possible solutions to these three problems known as the "two bits of wire" development system, the "MouseTrap" utility and the "Spooly" device are discussed below.

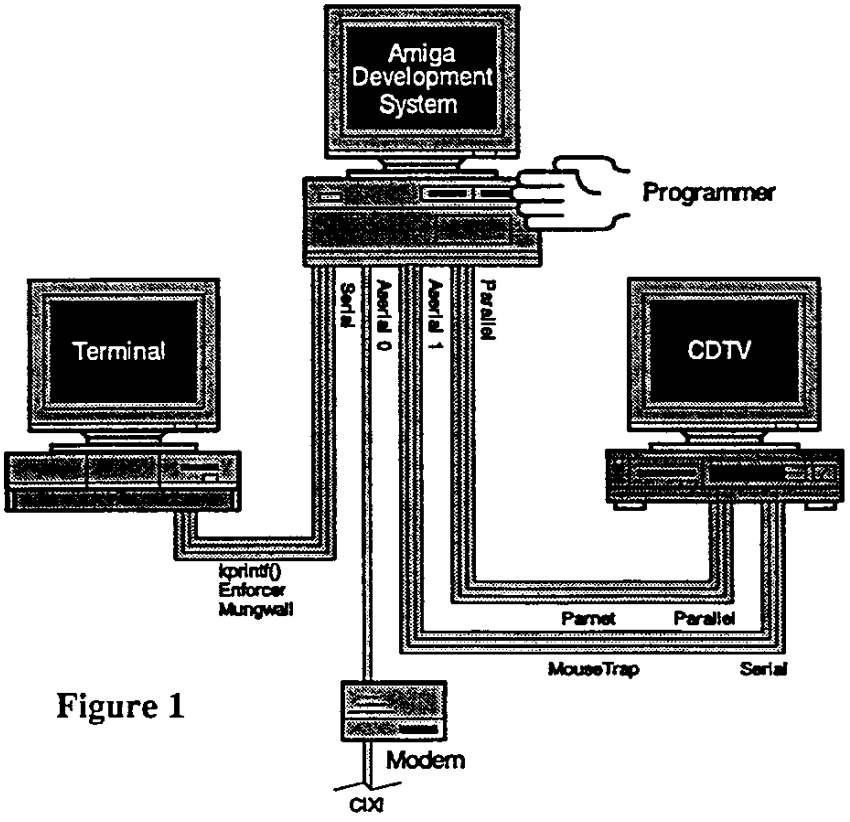


Figure 1

Three Screens Development System

A possible development system for CDTV programming is illustrated in figure 1. This is not intended as a complete multimedia development system, but as a comfortable (some would say luxurious) environment for a programmer. Different people like to work differently so there's no such thing as the "correct" development system, but it may contain features you might find useful. The use of three screens reflects the fact that multimedia programming is an information intensive business; with three screens you can have your output display, current source, and debug information always available.

Central to the whole thing is a high-end Amiga, running the fastest processor you can afford and a big fast hard disk. It's very convenient for this to use a multi-port card such as the Commodore multi-serial card or the Applied Systems "Amiox". The central system runs your authoring tools, compilers, assemblers, editors, etc.

The Diagnostic Terminal

To the left of the central development system in figure 1 is a diagnostic terminal, such as an old Amiga 1000 or 500 running a terminal program such as NComm. While this will be familiar to many people, it may be worth summarising three particularly useful sources of debug information:

- **kprintf()**
  A diagnostic printf() in debug.lib that outputs at 9600 baud down the serial port. There's a lot to be said for conditionally compiling diagnostics into your source, typically announcing the name of each routine as it is entered and the values it is being called with. This way, if the system crashes and the display bursts into flames, you can still find out which routine crashed and what it was trying to do.

- **Enforcer**
  A very useful CATS utility everyone should use. Enforcer uses the MMU on a machine with 68020 or above to catch illegal memory references (e.g., to location zero), and outputs diagnostic information about them to the serial port in a form known as "Enforcer hits". This helps catch a whole range of nasty C bugs involving NULL pointers and the like.

- **Mungwall**
  Another CATS utility, which catches buffer overflow conditions or attempts to use memory after it's been freed, by "munging" (scrambling) memory as soon as it is freed, and by putting a "wall" of special tokens

around allocated memory. This sort of error is responsible for a lot of intermittent bugs which are otherwise very hard to track down - using Mungwall in conjunction with Enforcer and kprintf() you can see exactly when this sort of thing happens, and which routine it happens in too.

In addition, there is the old remote monitor RomWack and the new one, SAD, plus a number of special remote debug facilities appearing in packages like *SAS C*.

Connection to CDTV - Two Bits of Wire

Moving to the other side of Figure 1, the connections between the development system and the CDTV make up what is known as the "two bits of wire" CDTV development environment. The bits of wire are as follows:

Serial Cable

This is a serial cable running between central system and CDTV serial ports (the multi- port card comes in handy here), running a freely-redistributable bit of software called MouseTrap in master/slave mode. This allows you to toggle the central mouse and keyboard between controlling the central system and controlling the CDTV, and also allows diagnostic information from the CDTV to be displayed on the central screen.

Parallel cable

This is a parallel cable running between central system and CDTV parallel ports, running a freely-redistributable bit of software called ParNet by the Software Distillery, allowing the CDTV to access files on the central system by means of the parallel port. A short floppy disk startup-sequence on the CDTV wakes up ParNet, assigns everything over to directories on the central system, then leaves the CDTV running entirely off the central hard disk. This means you can compile a program on the central system, then test it immediately on the CDTV across the net; it also means that if the program crashes by any chance, you can get on with bug hunting while the CDTV reboots.

ParNet on Workbench 2+

ParNet is quite old, and there's been some discussion on the nets as to whether it works correctly on Workbench 2. It appears that version on Fish 400 ("08/90 fixes for AmigaDOS 2.0") works fine, but gives some enforcer hits and suffers from a small memory leak. Used between a 2.0 or 3.0 system and a 1.3 system like CDTV it works okay as long as the CDTV is accessing 2+ resources; if the 2+ system tries to access CDTV resources, it doesn't work at all. Later versions may fix these problems, but for the two-bits-of-wire development system, they really aren't serious, as long as you remember always to access ParNet from the CDTV side.

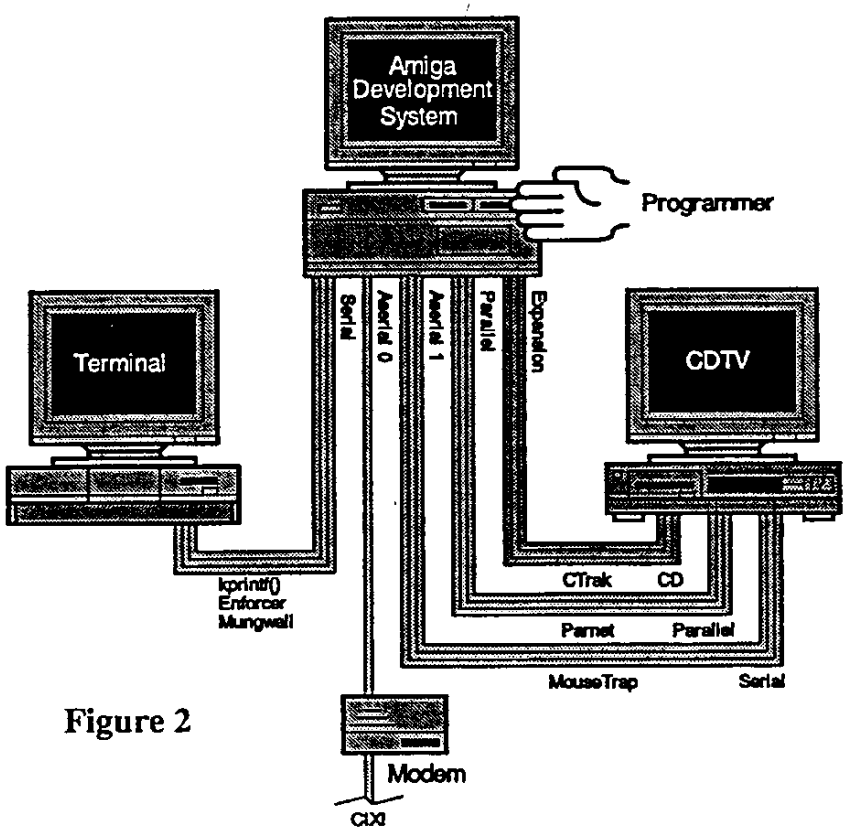


Figure 2

A Word About Emulators

While ParNet is fine for transferring test software, it's not much good for data - while it's faster than floppy, it's slower than CD, and you don't get anything like CD timing. So in order to use the two-bits-of-wire system, you really have to be prepared to cut an early gold disk with some test data, preferably in a variety of formats. You can then develop the software across ParNet, testing it "for real" on the CDTV as you go - a long way the safest option.

Figure 2 shows an alternative "luxury" option. This adds in a system called CTrac, which is a board plugging into the development system expansion bus, with a cable which plugs in place of the CD drive in the CDTV. This is a hardware emulator, allowing an ISOimage to be built on the development hard disk, then accessed from the CDTV with emulated CDTV timing. This provides maximum flexibility if you need to experiment with data formats etc., but like many good things in life it isn't cheap, plus you need a 68030 processor and fast SCSI controller to run it. An alternative is to start off running a software emulator such as Carl Sassenrath's SimCD on a partition on the central hard disk, then cut a test disk when you're ready to transfer to CDTV. It's also worth mentioning that since some work by Jim

Hawkins demonstrated at the last Buxton Devcon, it's now possible to drive a gold disk cutter directly off an Amiga at comparatively trivial expense up to audio data rates (!), so cutting an early data-only test disk should no longer be considered an expensive option.

Handling user-input with MouseTrap

Moving now to the second of the special problems of CDTV development, that of handling both the standard remote controller and wide variety of alternatives, it may be worth saying a little more about MouseTrap. The way this functions is illustrated in figures 3 and 4. Figure 3 illustrates the standard Intuition food chain, in which a high priority "input device" coordinates input events from mouse and keyboard (or IR controller on CDTV) and timer, and passes them to a chain of input handlers, the most important of which is usually the Intuition library. Figure 4 illustrates how this is modified by MouseTrap with a new process which handles getting new remote input events from the serial port for remote control, and a new high priority input handler, which does various diagnostic and input aliasing tricks. Tricks available from MouseTrap include the following:

- Operation between CDTV and Amiga in the master/slave mode, as already discussed.
- Output of diagnostic information regarding input events, including or

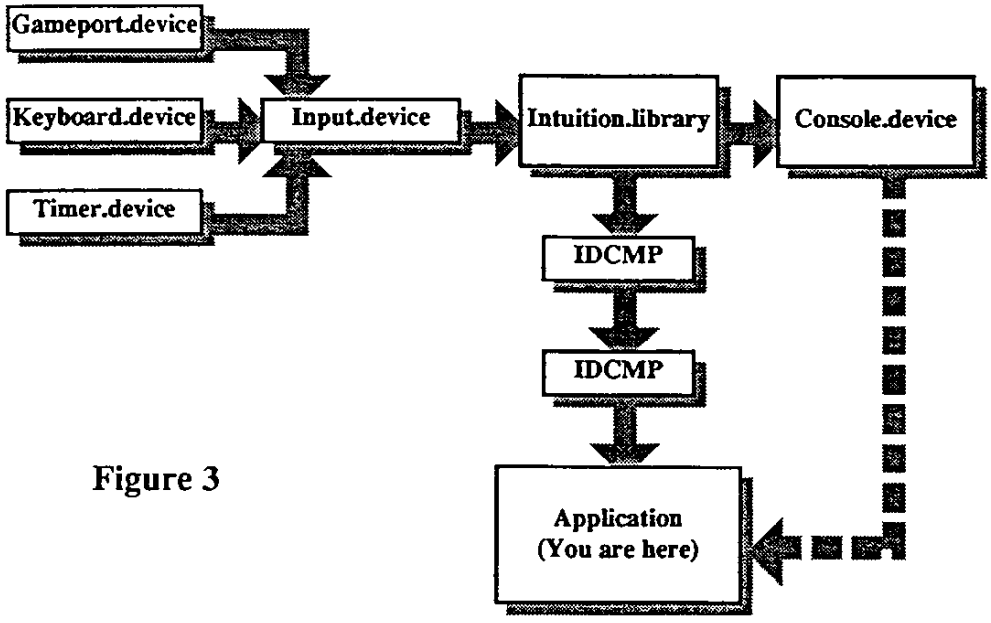


Figure 3

excluding timer events. This can be useful tracking down obscure bugs, including being able to tell if your software is missing input events, or if the IR controller is simply failing to generate them.

- "MouseTrap" mode in which mouse-move and button events are aliased to key events such as "U", "D", "L", "R" plus "A" and "B", or to cursor keys, with a specified constant repeat rate. This makes it much easier to implement the standard "bouncing highlight" user interface using the standard IR controller; it also helps avoid problems with the highlight zooming round the screen when a trackball is in use, by keeping the repeat rate constant.

- "Noball" mode, in which IR controller events are aliased to key events, but trackball and IR mouse events are left alone. This allows a user interface to adapt itself transparently to input device; if it's the standard IR controller use a bouncing highlight, otherwise put up a cross-hair or conventional mouse pointer.

MouseTrap is freely redistributable program, and comes with documentation and full source-code, including a short assembler routine "MiniTrap" for inclusion in other programs.

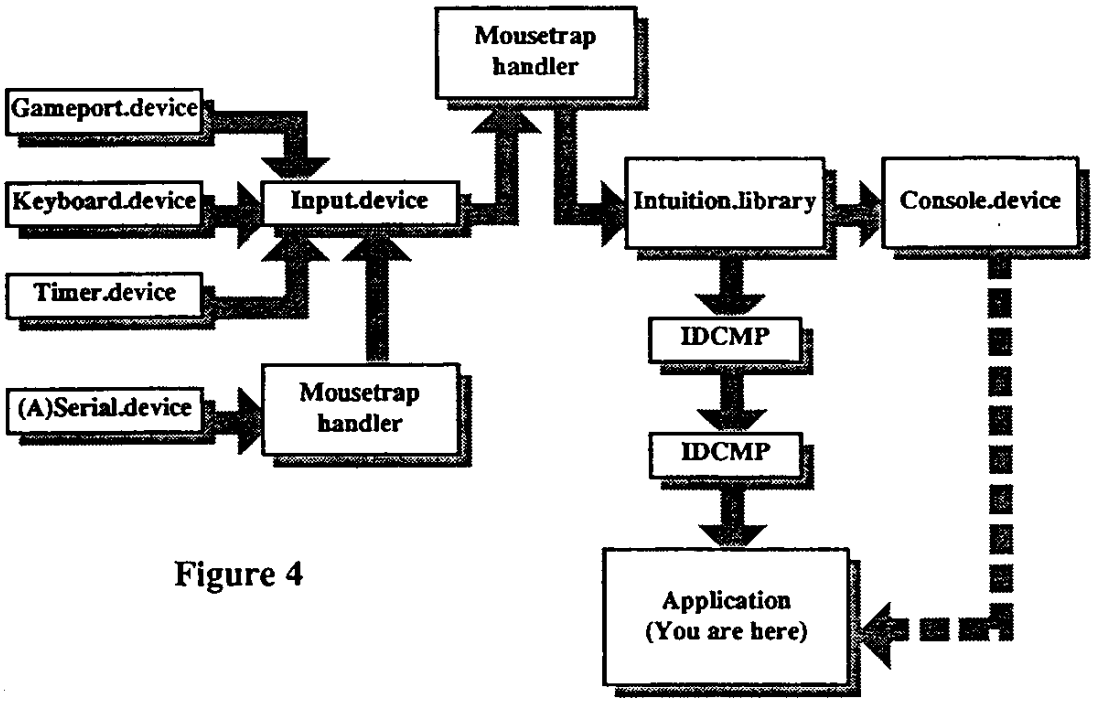


Figure 4

Avoiding wait-states - Spooly Device

Spooly.device is an attempt to tackle the third of the general CDTV development problems discussed above, and also to improve title performance generally, by spooling sound and animation material straight off the disk. This means that it should always be possible to respond to user input within one disk seek (under a second), and also that the length of sound and animation is limited only by total CD capacity, not by available RAM. In some ways it's not dissimilar to CDXL; however it differs as follows:

- It makes use of standard IFF 8SVX and ANIM5 forms, as output directly by many sound digitisers and animation packages.
- While CDXL provides true video quarter screen, Spooly fills the whole screen by cheating (i.e., not all the screen's updated at once).
- It turns out Spooly does nearly everything the opposite way to CDXL - but there's nothing to stop the two techniques being used side by side in the same title.

Spooly started life as an initiative by Jim MacKonochie of CBM(UK), and has subsequently been backed by Commodore International at West Chester.

How Spooly Functions

The original software structure of Spooly device is illustrated in figure 5. From an application program's point of view, Spooly appears either as a library, in which case it consists of a load of high-speed IFF decode and utility routines, or as a device, in which case it is capable of asynchronous I/O. An animation or sound reply can be started by a SendIO() call, stopped by AbortIO(), or you can wait for completion using WaitIO(). Since this is asynchronous, it's possible to get on with other things such as watching out for user input events while animation and sound are playing. All communication via the device interface is handled using a standard structure called a SpoolyRequest - it's designed to be as much as possible just like talking to any other device in the system. At the other end, Spooly gets things done by talking to audio device, graphics library, DOS library and CDTV device. Efforts have been made to keep Spooly as small as possible - it currently takes up under 30K of RAM when not active, plus if you close it when not in use, it will be flushed from the system if memory gets tight.

Functions of the individual device components are as follows:

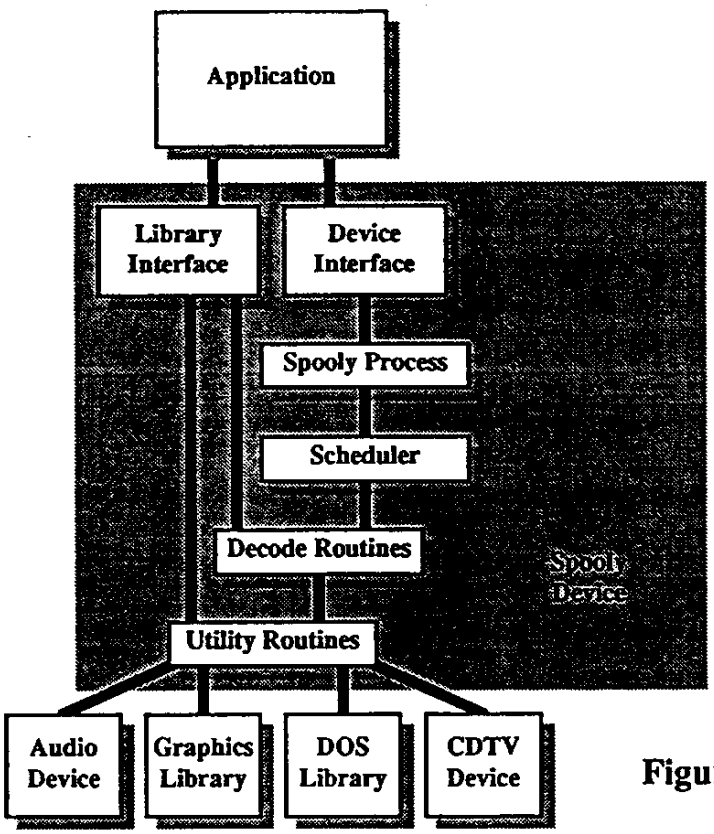


Figure 5

Utility routines

The utility routines are mainly concerned with getting maximum performance off the CD. They do this by worrying about whether it's more efficient to go through DOS, or to short-circuit DOS by going directly to cdtv.device for large transfers. They also worry about whether you're actually running off cdtv.device or not - if you are not, they'll just go automatically to DOS. An additional function of the utility routines is to handle directory buffering. If you have a variety of possible animations or sounds to select depending on a user-choice, and if these can be kept together in a small directory, then it's efficient to ask the system to get information about this directory in advance of the user-choice - the appropriate file can then be accessed as soon as possible after the user makes a decision.

Decode Routines

The decode routines are concerned with doing things like ILBM decode as fast as possible. It's conventional wisdom that you can't use things like ILBMs (or indeed ANIMs) on CDTV because the ILBMdecode time takes too long - several seconds in many cases. Spooly contains some painfully optimised assembler to get this decode time down to under a tenth of a second on an ordinary lowres five-bitplane ILBM, so this isn't really a big problem any longer.

Scheduler
The scheduler is the most complicated bit of the device. It deals with up to four audio channels and one "video channel" on the fly with various synchronisation tricks possible between them; it uses a multi-threaded structure with sound given priority, as any audio break-up is usually a lot more noticable than a pause in animation.

Spooly Process
The Spooly process is what handles asynchronous IO. It runs at higher than normal system priority (default 5) in order to keep things going as smoothly as possible, but leaves enough CPU to handle the user interface.

What you can do with Spooly

Called from a suitable application program, spooly.device is capable of the following:

- Called as a library, it can handle getting things into memory as quickly as possible, including directory pre-buffering. It can also do a very fast analysis of an ILBM in memory, and fast decode into a supplied bitmap.
- Called as a device, it can rapidly load and replay 8SVX sound or ANIM files from RAM, with various speed volume colour, etc., overrides, and options like animation loop and bounce. It can also handle decode of anim brushes into supplied backgrounds. Things like animation speed and colours can be changed on the fly in a playing ANIM with an appropriate device call. Mono and stereo sound samples are both supported. Animations can either be replayed into a supplied BitMap, or the device can create one for you; overscan is fully supported.
- Long mono sound samples can be spooled off hard disk or CD, giving almost unlimited replay time with minimum start delay. Long stereo 8SVX samples don't spool directly off CD very well as this involves a lot of seeking, though there's a trick to overcome this using "spool files".
- ANIM files can also be spooled off CD. In the simplest spool mode, animations are spooled a few frames at a time - this is very memory efficient, though only works well for animations with a limited amount of movement, such as characters moving against a static background. This limitation can also be overcome using spool files.
- Short sounds such as gunshots, etc., can be pre-loaded then synchronised with specified frames of animation replay.

- With a bit more effort, it is possible to replay a whole sequence of sound samples with associated animations, to provide a continuous synchronised sound and animation sequence. This involves some work outside the device - you have to maintain a circular list of SpoolyRequests, call Spooly as a library to load the corresponding data, then queue the replay requests for this data to Spooly's device interface.

Spool Files

Experience shows that while the last method mentioned above consumes the most memory, it also gives the best performance. It suffers from drawbacks however in that it involves rather a lot of work by the application programmer, and that the use of a whole series of synchronised sound and animation files can lead to a lot of seeking, which gives poor performance on CDTV. Spooly therefore now contains a facility to join such files together into a higher-level IFF known as a "spool file", and to replay from such a spool file without any extra effort by the application programmer, and without any seeking. The structure of a spool file is an IFF FORM containing a sequence of ILBM and ANIM FORMs interleaved with special chunks containing synchronisation information - see Spooly.doc for more information.

Replay of a spool file is illustrated in Figure 6. It can be seen the Spooly has created a new process to handle prefetch of data from the spool file; replay requests for this data are then queued to the old Spooly process. A number of special synchronisation facilities are available when using spool files. These include

- The ability to play animations and sounds until complete, or loop them for a specified time period.
- The ability to stop an animation as soon as its associated sound finishes. The animation may be run normally, looped, or "held" on its last frame.
- The ability to put an animation back to its first frame when sound finishes (useful for getting characters' mouths shut!).
- The ability to specify a given sound channel for prioritisation ("special sound synch"). The scheduler will then prioritise achieving smooth join-up of sounds on this channel (and any associated stereo pair) above all other activities.
- The ability to overlay IntuiText on part of the animated display.
- The ability to invoke user callbacks during display.

Spool files are currently created by a simple script-driven utility program, though it is hoped

to make something more elegant available in the future. The main use of spool files is to provide efficient synchronised sound an animation, but additional uses are the following:

- Chopping long animations into bits then replaying them via a spool file turns out to give much higher performance then replaying them a frame or so at a time using the simple spool option.
- Spool files also provide a way of spooling long stereo sound samples - a utility program is provided to split these into a series of shorter samples in a spool file.

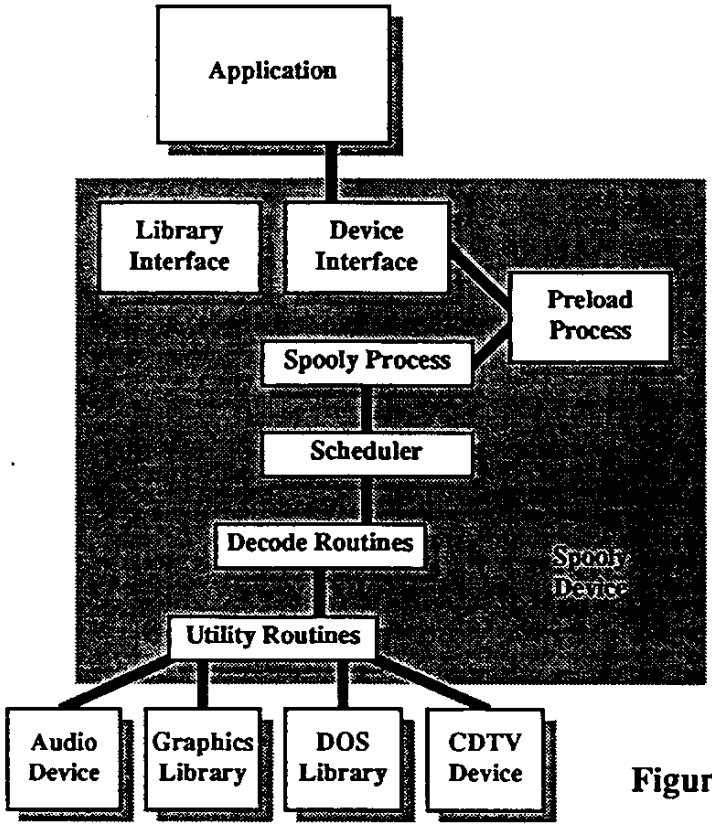


Figure 6

Accessing Spooly

As Spooly is an Exec device, it can be accessed from any programming system capable of talking to such things, including assembler, C, Modula 2, and high speed Pascal. Opening the device is absolutely standard:

```
#include "spooly.h"
struct MsgPort *SpoolyPort;
struct SpoolyRequest *SpoolyReq;

if (!(SpoolyPort = CreatePort(0,0)))
       ErrorExit("Can't create port");

if (!(SpoolyReq = CreateExtIO(SpoolyPort,sizeof(struct SpoolyRequest))))
       ErrorExit("Can't create IO request");

if (OpenDevice("spooly.device",0,(struct IORequest*)SpoolyReq,0))
       ErrorExit("Can't open device");
```

plus if library access is required the base address must also be picked up:

```
struct SpoolyBase *SpoolyBase;
SpoolyBase = (struct SpoolyBase *)SReq1->Req.io_Device;
```

Performing device commands is then a question of filling in the appropriate commands in SpoolyReq structures, as will be found fully documented in Spooly.doc. For example, to spool an animation:

```
SpoolyReq->Req.io_Command = SPCMD_SHOW;
SpoolyReq->Name = "MyAnim";
SpoolyReq->Id = IFF_ANIM;
SpoolyReq->Flags = ANIMF_SPOOL;
DoIO((struct IORequest *)SpoolyReq);
```

It is to be hoped that writers of authoring systems will in the fullness of time provide links to Spooly, but in the meantime there are two other options:

- A short program AShow exists to invoke Spooly from a DOS command. This can be run from anything which is capable of executing a DOS command, and can also be made resident to avoid having to reload. Source-code for AShow is available as an example of how to use Spooly.
- A special extension is being built to talk to Spooly from Amos.

References

Enforcer and Mungwall are distributed by CATS(USA). The latest Enforcer by Michael Sinz is a complete recode - see the doc files for details.

ParNet is available on the Fish disks - version 08/90 is on Fish 400 with documentation.

MouseTrap is freely redistributable, and can be found with documentation on CIX and on the Almathera CDPD II CD (which also has ParNet) amongst other places. Handling alternative input devices is also discussed at some length in a section of the CDTV Developers Reference Manual (updated version).

Spooly Device is available from CATS(USA). For full information on all device and library calls see Spooly.doc

◆

ISOCD and OptCD & SimCD

Copyright

Warning

The information contained herein is subject to change without notice. Commodore specifically does not make any endorsement or representation with respect to the use, results, or performance of the information (including without limitation its capabilities, appropriateness, reliability, currentness, or availability).

Disclaimer

This information is provided "as is" without any warranty of any kind, either express or implied. The entire risk as to the use of this information is assumed by the user. In no event will Commodore or its affiliated companies be liable for any damages, direct, indirect, incidental, special, or consequential, resulting from any defect in the information, even if advised of the possibility of such damages Important

Note

The tools ISOCD, OptCD, and SimCD are available only to Commodore-licensed CD developers. They are documented here solely as a reference aid, and are not included in the Devcon electronic materials.

For more information about becoming a licensed CD developer, please contact CATS administration.

ISOCD and OptCD

Table of Contents

SimCD

Table of Contents

ISOCD Overview

ISOCD will examine a directory or partition and create an ISO-9660 image, ready to be mastered into a CDTV disc. The ISO-9660 image will have the necessary data to be used on ISO-9660 and the disc produced can be used on other systems, aside from the Amiga, such as Macintosh, Unix, or IBM compatible computers. Besides CDTV, the disc is usable by the existing CD-ROM file systems for the Amiga. Most importantly, the SimCD simulator can treat a partition created with ISOCD as if it were the CD0: drive of a CDTV.

ISOCD has several important features:

- You can graphically re-arrange the actual order of files, changing how they will be accessed. This will greatly improve the access times on CD-ROM drives.
- These layout files can be re-used.
- It uses a minimal amount of memory for the image creation.
- It allows the setup of custom CDTV CDFS options.
- Files can be grouped and ordered automatically.
- ISOCD can optimize the access order of your files and directories based on information gathered by OptCD. OptCD monitors CD0: access while you are walking your application through its paces.
- The ISO-9660 headers can be produced separately to allow a small initial data track for CD-ROMs that are audio compatible.
- The layout file is text based so it can be manipulated by other tools of your own.

Usage philosophy

You can use ISOCD to make ISO-9660 image files to be sent off and made into CDTV CD-ROMs. With SimCD and a large enough partition on your hard drive, you can use ISOCD to create simulated discs for testing your application without actually pressing discs! Of course, final testing should be done with actual CD-ROMs on a CDTV, but most of the development cycle can occur without the hassle and expense of outside disc pressing. Actual placement of the files and directories on a CD-ROM is very important for maximum performance of your application on the SLOW seeking CD-ROM drives. Because of this, ISOCD allows you to rearrange the order of files and the like. These ordered listings can be saved and loaded as layout files.

CDTV.TM

CDTV.TM is provided with ISOCD as the trademark file for CDTV discs. This file allows the application to run on a CDTV. ISOCD will place this file close to the beginning of the

CD-ROM since it is one of the first files read at boot time. If this file is present in the current directory, it does not need to be present in the source directories. If properly found, it will show up in the examined list as <Trademark>.

Memory/resource usage

Run time memory usage of ISOCD is strictly dependent on the number of files and directories in your application. To figure rough memory usage, take the size of internal buffer, add 140K and an average of 50 bytes per file or directory entry. The internal buffer is normally 256K in size, refer to -f under Batch for changing this size. So an application with 5,000 files and directories and a default buffer would take approximately 650K of memory to build.

Examples of use

To help you get started quickly, let's do a very simple ISO build right now.

Follow these steps:

- Start up ISOCD.
- In the Files box, click on [Source],
- Select the "S:" directory, click [OK].
- In the Files box, click on [Image].
- Enter "RAM:Test.iso," click [OK].
- In the Actions box, click on [Examine].
- In the Actions box, click on [Build].
- That's it, "RAM:Test.iso" is an ISO-9660 image ready to be made into a disc.

General procedures

Usage of ISOCD is rather simple. The following steps represent a simple build:

- Check that the options are set correctly.
- Set the Source directory.
- Set the Image to a file or partition.
- Examine.
- Rearrange files as desired.
- Save the layout file if desired.
- Build.

After an examine, you can save the list of files out as a layout file. This file can be externally edited as to the order of the files and reloaded or you can use the editor within ISOCD. Later usage on the same project can use the layout file so that you do not have to re-order anything but the new files.

Philosophy

The ability to re-arrange the layout of files in the image is important to optimizing the execution timing of an application. The directory structure of the source material is always preserved, but the physical layout of files is completely independent. This means that the files used first on booting should probably be at the head of the list, closer to the center of the CD-ROM. This minimizes the deadly seek time inherent to CD-ROM technology. If the files are arranged properly, a great number of seconds can be shaved from the load time of your title. A little thought on the physical layout of files can actually create a responsive title - one that actually fits the needs of the consumer.

System Setup

ISO-9660 images or partitions can be very large. It is wise to have more than twice as much hard drive space as your application actually uses. It is even more wise, in fact almost Guru thinking, to use a separate drive for the image. This reduces the risk involved with such large projects. ISOCD is very meager with its memory usage, but if there are a tremendous number of source files, you should have at least 2 megabytes of memory.

In case you feel overloaded with all of the files needed for a multimedia application, consider Hypermedia's Fred Fish CD-ROM. The version 1.5 disc created had almost 600 megabytes and 67,000+ files! ISOCD built the image in just a little over two hours. The Fred Fish Online CD-ROM was 415 megabytes and 4,000 files. It took ISOCD 16 minutes. Of course, your performance will vary according to disk speeds, DOS fragmentation, etc.

Options

Options are saved with the layout file. They control the examine process, the header of the ISO-9660 image, and the interface to the CDFS in CDTV during the boot process. Directory ordering and grouping must be set before an examine, and the other options must be set before a build.

ISO
ISO-9660 provides an area in the header of an image called the Primary Volume Descriptor, or PVD. Here you specify the title, publisher, etc.

Dir Order/Rev
The directory order present on the original source is always preserved. ISO-9660 specifies the ordering of files and directories to be alphabetical. But we are free to place the physical locations of the files and directory entries. When examining and building the ordered list of files, ISOCD can perform automatic arrangement of files. Under Dir Order, you can specify

file arrangement within a directory alphabetically, by size, date, or how they are originally ordered. Order is from A-Z, least to largest, or oldest to most recent. This can be reversed. Of course, you are free to arrange the files after an examine.

Remember that this is the physical position on the CD-ROM, not the order "LIST #?" would produce, since LIST uses the ISO-9660 directory structure to report files. Also note that these orderings are per directory, not over the entire CD-ROM.

Dir Group/Rev

In conjuction with ordering, you can group files together within a directory. You can group ".Info" files, directories, common file extensions, or not at all. Grouping directories is the most useful since the CDFS uses a directory cache system that may be able to load all of the current directories in one read if they are grouped together. This is obviously application dependent.

Volume ID

Volume ID is the CD-ROM title, 32 characters maximum. The remaining descriptors are limited to 128 characters.

Volume Set

Place the volume set name here.

Volume Pub

The publisher is specified here.

Volume Prep

Indicates the actual preparer of the data.

Volume App

Indicates the application name.

PVD

Though only one PVD is needed for a CD-ROM, you can specify more than one for data redundancy to improve reliability. CDTV will try another PVD if there is an error reading one.

Base Sector
Split File

Base Sector and Split File provide the ability to place the bulk of the application in a different track than the first. You specify a position for the data track in the Base Sector and a file for the header. This gives you two data tracks, one which is the header and must be placed at the beginning of the CD-ROM, and another that can be placed after audio tracks. This renders a CD-ROM that the consumer can use as an audio CD and it really is a CDTV disc as well!

CDFS

The CDFS in CDTV allows custom configuration of many variables through a data area in the PVD. You can set them here to tailor CDTV to your application.

Data Cache

CDFS has a cache for reads, specified in number of sectors. Keep in mind that each sector cached represents 2048 bytes of memory. There can be from 1 to 127 sectors cached. The default value is 8 sectors.

Dir Cache

More important, CDFS caches directories. You can specify from 2 to 127 sectors to be cached, each using 2048 bytes. On a CD-ROM, there is always seek timing to be considered. If all directory entries for an application can be read into the cache in one read, the savings in access time is tremendous. Properly used, the CD-ROM can be almost as responsive as a hard disk. Again, this uses memory that cannot be recovered by the application. The default value is 16 sectors.

File Lock

File locks are managed in pools, from 1 to 9999 locks, each using 32 bytes. The default value is 40 locks.

File Handle

File handles are also managed in pools, from 1 to 9999 handles, each using 24 bytes. The default value is 16 handles.

Retries

You can specify from 0 to 9999 retry attempts on read failures. The default value is 32.

Direct Read

This option allows an application to wring the ultimate in performance out of the original CDTV. CDTV has a bug in the CD-ROM drive hardware that occasionally will not transfer the last few bytes. The system handles this for all applications, but at a performance penalty. There are three ways to read data at the optimal, unprotected, rate:

- Use the cdtv.device directly with the CDTV_READ or CDTV_READXL commands.
- Turn on Direct Read for an opened file by sending a ACTION_DIRECT_READ packet for the file.
- Turn on Direct Read for the entire CD-ROM, with this option.

When reading directly, you must pad your buffers and your disc data by the valueREAD_PAD_BYTES, and request READ_PAD_BYTES more bytes than you would otherwise. Thus, any missing bytes are part of the padding, and ignored. ACTION_DIRECT_READ is defined in the CDTV manual. READ_PAD_BYTES is defined in <devices/cdtv.h>.

You must understand that normal workbench commands, LoadSeg, and other programs *do not* know to handle their reads differently with Direct Read on for the entire CD-ROM. This means writing your own custom loader and executing only your programs on a CD-ROM with Direct Read set. This also means, for most practical purposes, that this option is not really usable. We still *highly* recommend using CDXL (CDTV_READXL) and direct CDTV_READs within your application in any case. It is easier and a lot faster!

Any new CDTV machines or CD-ROM drives will not suffer under this curse.

Trademark

The Trademark file is necessary for the application CD-ROM to boot under CDTV. It is removed from the screen with RMTM on booting, please refer to the CDTV documentation. ISOCD places this file outside of the directory at the very inside of the CD-ROM for faster booting. The file CDTV.TM is the Trademark file. If you place it in the current directory when you run ISOCD, it does not need to be in the source directories. This option defaults to on.

Fast Search

Fast Search is an optimization available if the volume is prepared with directories that are sorted case insensitive. ISOCD works this way so it is normally on. This optimization enables CDFS to stop its search for a file in a directory if it passes the position alphabetically.

Speed Independent

CDTV uses a CD-ROM drive that reads in data at 75 sectors a second, or 150K/second. As we all know, faster drives are being produced. This option lets any future machine know that the application would be more than happy with faster reads. This is up to you, we will not twist your arm.

Of course, all of the caveats of good programming apply if you want to be speed independent. The 75hz timer within CDTV will always be available for easy timing control, independent of NTSC or PAL. We recommend you use this, it is CDTV_FRAMECALL with cdtv.device.

This option defaults to off.

CDFS Options

Option	Default	Minimum	Maximum
Data Cache	8	1	127
Dir Cache	16	2	127
File Lock	40	1	9999
File Handle	16	1	9999
Retries	32	0	9999
Direct Read	OFF		
Trademark	ON		
Fast Search	ON		
Speed Ind.	ON		

Restore Defaults
This resets all of the CDFS options to their defaults. It is available for those times when real confusion sets in.

Load/Save Layout

Layout files contain all of the information necessary to build an ISO-9660 mage. Even though ISOCD lets you edit this list, it can also be edited on the outside by any TEXT editor, not a word processor, unless it can save as a text file. ISOCD will not understand any particular word processor format or special codes.

Here is an example layout file. It's a 1.3 devs directory examined using alpha order and dir grouping.

```
0 0 1 0
0 0 1 0
2 1024
8 16 40 16 32
0 1 1 0
CDTV_TEST
Test Set 1 of 4
Pantaray, Inc.
Kenneth Yeast
Application Name
TestHeader.iso
Test.iso
0004   Systeml.3:devs
0001   <Root Dir>
0001   clipboards
0001   keymaps
0001   printers

H0000  <ISO Header>
P0000  <Path Table>
P0000  <Path Table>
C0000  <Trademark>
D0001  <Root Dir>
D0002  clipboards
D0003  keymaps
D0004  printers
F0001  clipboard.device
F0001  kickstart
F0001  MountList
F0001  narrator.device
F0001  parallel.device
F0001  printer.device
F0001  ramdrive.device
F0001  serial.device
F0001  system-configuration
F0003  cdn
F0003  ch1
F0003  ch2
F0003  d
F0003  dk
F0003  e
F0003  f
F0003  gb
F0003  i
F0003  is
F0003  n
F0003  s
F0003  usa0
F0003  usa1
F0003  usa2
F0004  Alphacom_Alphapro_101
F0004  Brother_HR-15XL
F0004  CalComp_ColorMaster
F0004  CalComp_ColorMaster2
F0004  Canon_PJ-1080A
F0004  CBM_MPS1000
F0004  Diablo_630
F0004  Diablo_Advantage_D25
F0004  Diablo_C-150
F0004  EpsonQ
F0004  EpsonXOld
F0004  EpsonX[CBM_MPS-1250
F0004  generic
F0004  Howtek_Pixelmaster
F0004  HP_DeskJet
F0004  HP_LaserJet
F0004  HP_PaintJet
F0004  HP_ThinkJet
F0004  ImagewriterII
F0004  Nec_Pinwriter
F0004  Okidata_293I
F0004  Okidata_92
F0004  Okimate_20
F0004  Quadram_QuadJet
F0004  Qume_LetterPro_20
F0004  Toshiba_P351C
F0004  Toshiba_P351SX
F0004  Xerox_4020
E0000  65536
```

The layout file is a text file that contains the options, directory structure, and editable order list. The first two parts of the file are *not* to be changed, this would produce unpredictable results. The first part of the file is 11 lines of option data. The next section is a listing of the directory structure, terminated with a blank line. The third section is yours to rearrange, it is the order of files/dirs/etc., on the CD-ROM.

The actual text in each of the lines should *not* be changed, only the order of the text lines is changeable. Also, no line should be deleted or added. This should only occur within ISOCD.

Once you have your order figured out and saved in a layout file, you can automate the build process during development. Remember to specify the image file or partition before saving the layout file. Assuming your layout file is called Test.lay, simply call ISOCD as follows:

```
ISOCD -lTest.lay -b
```

This would build an image without any interaction, allowing its usage in an arexx script or batch file.

Examine

This will examine the source directory and build the list necessary to create an ISO-9660 image. You can alternately load a layout file from a previous examine. The examine will take into account the directory order and grouping specified in the options. The examine will display each directory that it is loading and you can abort at any time. An examine will clear any previous list in memory.

After an examine, Status will report the number of bytes in the image, the number of directories, and the number of files. This number of bytes is usable in creating a TOC file, assuming you have audio tracks as well.

The ISO-9660 standard specifies that directories cannot be greater than eight deep. ISOCD will allow any number deep, after all - this is the Amiga - but will report if you have violated the standard.

Optimize

Optimize will take data accumulated in a statistics file by OptCD and use the totals to change the order of files and directories in your layout. Despite the type of data collected, you have the choice of what will be used in optimizing the list. If you wish to re-arrange only directories, turn on the Directory No Cache and Cache only. If you wish to optimize files only, use File only. Normally you will have no need for Block or Low Level Read, but it is here for completeness.

You can only optimize a layout once, since the statistics file is based on the existing CD layout only. If you wish to try again, simply load the layout file again.

> *Note* that all reads that are used are given equal priority. You may wish to optimize directories only at first, then optimize files after a rebuild and another OptCD monitoring.

Statistics

This is the file produced by OptCD when monitoring your application. See OptCD. It provides the access data and is only valid when paired with the layout that exactly matches the CD0: image.

Directory No Cache
Directory Cache
Note directory reads that were or were not in the directory cache.

Block No Cache
Block Cache
File reads and directory reads that are not in the directory cache, use the block reads. This notes those block reads in or out of the cache.

File
All file reads will be noted.

Low Level Read
Note any read of the CD.

Method
Float Most Read to Top
This will move the most read to the top after <Root Dir>.

Group Entries by Access Order
This method is more complicated. It notes when one file or directory is read after another. Multiple occurences of a pairing give it a higher priority. The highest pairings are done first, then the next, and so on. If an entry is to be moved after another and it is already paired, it will not be moved. However, if it has one paired with it, they both will be moved, etc. It is important that you walk through your application with the order in mind when you plan to use this method.

This method works best when used for files only.

Float Dirs to Top - No File
This is provided if you wish to simply move all directories after <Root Dir> at the top. If you have just a few directories, you may wish to fit all of them into the directory cache and never require a CD read for directories ever again, see CDFS under Options.

Mark What Was Changed
This will highlight the files or directories moved during optimize. In this way you can see what was changed.

Optimize
Perform the optimization. Remember that you can try another method by loading the layout file again.

Build

Build actually creates the ISO-9660 image file or partition. Remember that an image file can be very large. If an image is written to a partition, then that partition can then be simulated as CD0: or as just an ISO-9660 volume with SimCD. Refer to the SimCD documentation on the usage of CDFS and the simulator. While building, ISOCD will display a progress bar and can be aborted at any time.

The progress bar does not show writing of the directory blocks, so if you have all of your directories at the beginning of a list, there will be a small delay before the progress bar registers.

> *Warning - Warning - Warning*
> Please exercise unbelievable caution when using build on a partition. Once started, it will obliterate anything that was previously on that partition. Use it as you would use the DOS format command.

When you specify an image, you can enter a file name or a partition, such as CDN:. Because the file requester is dual purpose, entering a partition that is not an AmigaDos partition in the Drawer text will cause the requester to attempt reading of the partition. Of course, since it is not able to read the files in that partition, the requester will report an error. Simply choose Cancel, then OK in the requestor. Or, as we have found, enter the partition name, such as CDN:, in the File text.

If you are using the simulator, remember to write to the partition that is being simulated, such as CDN:. CD0: is a read only partition based on the ISO-9660 partition and cannot be written over. ISOCD will diskchange the partition being built and CD0: if it exists, but ISOCD does not know about any other devices that might be simulated as well from the image partition. You must remember to diskchange these yourself.

Files

Source

This sets the source directory to be examined. If you have previously examined, changing the source will clear the list. All files and directories in the source will be examined.

If you do not specify a source directory on the command line, it will default to the current directory. Remember that the layout file stores all of the information, so loading a layout file will set the source directory to the one for that list.

Image

When you specify an image, you can enter a file name or a partition, such as CDN:. Because the file requester is dual purpose, entering a partition that is not an AmigaDos partition in the Drawer text will cause the requester to attempt reading of the partition. Of course, since it is not able to read the files in that partition, the requester will report an error. Simply choose Cancel, then OK in the requestor. Or, as we have found, enter the partition name, such as CDN:, in the File text.

Layout

This sets the layout file name, refer to Load/Save Layout for more information.

Path/File

This gadget shows you the list of all entries to be written to the ISO-9660 image. It also allows you to re-arrange the order of everything, except for the ISO-9660 headers, always at the beginning, and empty space at the end of the image.

To move a file or set of files: first click on all of those to be moved, then, while holding down the Ctrl key, click where you want them to go. ISOCD will move the marked entries after the entry clicked on with Ctrl. If you wish to use Shift or Alt, instead of Ctrl, specify -q1 for Shift, or -q2 for Alt on the command line. You can also select an entire range by simply holding down the mouse and moving over the entries. The list will scroll for you, accelerating if the mouse continues moving outside the gadget. To deselect an entry, click on it again; extended select works in reverse as well.

ISO-9660 deals with everything in sectors of 2048 bytes. All files consume an even number of sectors; this means a file of 920 bytes uses 2048 bytes on the CD-ROM. This applies to all items on a CD-ROM, such as Path tables or directories. ISOCD displays the file sizes as they originally are, or there is a menu option that lets you display the sizes rounded to the sectors occupied or even dates instead. Amiga-S sets ISOCD to display actual sizes, Amiga-E sets to rounded sizes, and Amiga-D sets to display dates.

Dates are displayed like the LIST command, except for "today," etc.

The Path number is an ISO-9660 convention. For entries such as Path Table or Empty Space, they have no meaning. But for a directory, it is the directory number and for files, the directory they belong under. This preserves the directory structure since ISOCD internally remembers what directory a directory belongs under. In other words, if a given directory has a path number of 12, any file with a path of 12 is under that directory.

Besides directories and files, there are five other entry types.

- The ISO-9660 Header is shown for completeness, since you cannot move it. It encapsulates the 16 blank sectors, the PVDs, and the terminating volume descriptor.
- The Path Tables provide a short cut for accessing the CD-ROM. A second one is provided for reverse byte systems, such as Intel based computers. This can be removed by -p on the command line. We must caution you that many disc mastering systems require the extra path table in order to press the CD-ROMs.
- The Trademark file is necessary for your application to run on a CDTV. It does not need to be in the list of files in your source directory if it is in the current directory. The Trademark file is called CDTV.TM and is provided with ISOCD.
- The Root Dir is the root directory of the CD-ROM.
- Empty Space is provided at the end of the volume because older CDTV units could experience problems if someone attempted reading past the end of a CD-ROM. Empty Spaces can also be added to other places, refer to Add/Remove Spaces below.

You will notice that the Path Tables, Trademark, and Root Dir are placed at the beginning of the list. This improves boot time and running performance. It is also a wonderful idea to hand place the files used at boot time in order of usage at the beginning of the list, after any directories that need to be cached at the beginning. This takes a certain amount of thinking about how your application starts and might lead you to try some different philosophies. What works on a hard drive, such as echo commands and the like, are ridiculous on a CD-ROM. Consider just one program in your startup-sequence that handles everything possible. Remember that CD-ROMs are not changeable by the user, so a lot of the methods that are necessary for a changing environment such as a hard disk, are not useful on a CD-ROM.

The layout file will contain your arrangement of files so that you do not have to re-arrange entries again. If you are loading an older layout file for a source directory and there are new files or some have been deleted, ISOCD will compensate. New files will be placed at the end, you will need to remember to move them to where you would like. The actual directory structure cannot change, as this would changed all of the path numbers, described above. If you have added or removed any directories, you will need to examine again.

The layout file can be externally edited by your favorite text editor if you wish. Refer to Load/Save Layout above for more information.

Status

The first line reports the size of the ISO-9660 image, number of directories, and number of files. The second line reports any usage errors, refer to Status Errors.

Menus

- **Amiga-? "About..."**
  About box.
- **Amiga-Q "Quit"**
  Quit the program. It is also available on some of the little windows to leave the window.
- **Amiga-S "Display Sizes"**
  Display the size of entries, such as file sizes or directory sizes.
- **Amiga-E "Display Sizes Rounded"**
  Round the entry sizes to the actual space occupied on the CD-ROM, since the CD-ROM uses sectors of 2048 bytes.
- **Amiga-D "Display Dates"**
  Display dates in the LIST format instead of sizes.
- **Amiga-F "Find File..."**
  This allows you to jump to a given entry in the list, useful if you have a large set of files. It uses the AmigaDos search pattern, case insensitive, just like dir. Next and Prev is available to continue the search.
- **Menu items available:**
  Amiga-N "Next"
  Amiga-P "Prev"
  Amiga-H "Help"
  Amiga-Q "Done"
- **Amiga-A "Add Spaces..."**
  ISOCD provides a special capability for future use by allowing blank spaces within an ISO-9660 image. Adding spaces after certain files would allow someone to update just that file within an image being used for simulation. In this way, someone could just update the size field in the directory and rewrite a new version of a file instead of going through an entire build. A future utility might just provide this capability. leave the window.

To add spaces, simply mark those files that need spaces, choose this option, and specify the sizes. If no files were marked, one empty space would be added to the beginning of the list to be moved later. The files marked would be unmarked after an add.

- **Amiga-R "Remove Spaces"**
  Likewise, mark those Empty Spaces to be removed, then choose this option. If none is marked, ISOCD will remove the first available, keeping in mind that one is always needed at the end of the CD-ROM.

Batch

USAGE: ISOCD [options] <directory>

-l<file>	Layout file
-o<file>	ISO-9660 output file or device
-s<file>	Split ISO descriptor file
-a<file>	Statistics file for Optimize
-b	Build image (requires layout file (-l), will exit when done)
	NOTE: Use this option with the greatest of care...
-x	Verbose debugging information
-f<n>	# of sectors in file buffer [128]
-m	Use mimimum size window
-q<n>	Change qualifier for insert in lists [0]
	(0=Ctrl, 1=Shift, 2=Alt)
-t	Do not use topaz font, use system font instead
-p	Do not add Path Table for reverse byte systems
-v	Do not append version number ";1" to file names

Options:

<directory>

This is the directory or partition to be examined. If not specified, ISOCD will assume current directory.

-l<file>

Load layout file previously saved by ISOCD. ISOCD will ignore any missing files from the layout file list. Any new files will be appended to the end of the list. The directory structure cannot change however, there cannot be any missing or new directories in the list. This changes the directory numbering scheme of ISO-9660 such that ISOCD would not be able to reconfigure a given list. If directories themselves have changed, re-examine and re-order the list. This option is often used with -b to automatically build after loading the layout list. In this fashion, builds can occur within batch file or arexx scripts, automating program simulations. Layout files specify the build image within the file, so the -o<file> option is ignored.

Example:

```
ISOCD -lTest.lay -b
```

-o<file>

This is the output image file or partition to be built. Building to a partition allows you to use SimCD and simulate that partition as CD0: after a build. Remember to diskchange any dependent devices after a build, see Build. Please use great care when writing to a partition. ISOCD will

blindly trust that you are prepared to obliterate that partition with an ISO-9660 image. If the partition has not been used for an ISO-9660 image before, ISOCD will ask before proceeding with the build. But it will ask only once. The build process should be used by professionals, do not try this at home.

-s<file>
ISOCD can create a separate header image file. Please refer to Split File under Options.

-a<file>
Statistics file for Optimize function. This file is created by OptCD while monitoring your application. Refer to Optimize and OptCD.

-b
This will automatically build an image file if you have specified a layout file with -l<file>. This option is commonly used within a batch or arexx script, especially when doing repetitive testing with the simulator. Again, it cannot be stressed enough, please use this with *great care*.

-x
This turns on some primitive and profuse debugging text. If something is not happening correctly, this will help to locate where things go wrong. I assume that you are using enforcer, mungwall, io_torture, etc., for your good health and mine.

-f<n>
ISOCD uses a 256K buffer, 128 - 2K sectors, for its file transfers. Performance can be improved with larger buffers, if desired.

-m
ISOCD asks for a fat window when running so that the file list is as large as possible. -m will use a minimum size window for simpler usage.

-q<n>
When specifying where a group of marked files are to be moved to in the file list, you click where they are to go while holding down the Ctrl key. This allows you to use another modifier key, such as Shift or Alt.

-t
Since the list of files requires a fixed width font, ISOCD uses the Topaz font. If your system font is not too large and is fixed size, specify -t and ISOCD will use it instead.

-p

ISOCD will automatically include a path table entry for the "other" computer systems that use backwards ordering of bytes. This is often necessary for even CDTV specific titles, since many mastering systems are done on these "other" systems and they often need to look at the image. If you really do not want this extra path table, just say no with -p.

-v

In a similiar vein, many file systems require the ";1" version strings on all files. CDTV will handle file names with or without these strings, but you can turn them off with -v. Currently there are no provisions for actually specifying other versions instead of ";1".

Status Errors

"Need to Examine first"

A given function can only work if you have done an examine first to provide a list of files/dirs/etc. You cannot build until you have something to build, etc.

"Too many dirs deep for ISO"

This one reports that the source examined happens to be more than eight directories deep. ISO-9660 standard does not accept this, though CDTV and ISOCD will.

"Incompatible dir heirarchy"

When loading a layout file, there cannot be any additional or missing directories. If there are, the load will not complete.

"No more Empty Spaces to remove"

You are attempting to remove an Empty Space when there are none available to remove. There must always be the one at the end of the CD-ROM.

General Errors

"Batch option requires a layout file"

If you specify -b to build automatically, you must also specify -l<file> to load a layout file to be built.

"Not an Amiga system"

Failed to open graphics.library.

"Intuition (V2.0) not available"

ISOCD requires v2.0 of intuition or greater.

"Could not get current directory"
This error should not occur.

"Interface not available"
Probably not enough memory for window.

"Could not create gadgets"
Probably not enough memory for gadgets, or you are using -t and your system font is too large.

"No Memory"
Not enough memory.

Glossary

CD0
The CD-ROM drive on a CDTV or the partition simulated as this drive with SimCD.

CDFS
The CDTV CD File System. This allows you to treat CD0: as just another Amiga drive, even though it is in ISO-9660 format.

cdtv.device
The device that controls the CDTV hardware or the simulation of it with SimCD.

CDTV.TM
The trademark file that allows an application to run on CDTV.

CDXL
The fastest way to transfer data from the CD-ROM drive into the CDTV. CDXL basically starts a read and begins transferring the incoming data into places in memory that you specify. CDXL nodes control the shotgun effect of the data being read in, since the read does *not* pause.

ISO-9660
Refers to the 1988 ISO-9660 standard, specifically ISO 9660: 1988 (E), corrected 1988-09-01.

Layout Files
These text files store the information necessary to build an ISO-9660 image, including the order of files and directories.

PVD
Primary Volume Descriptor. This is the header for the CD-ROM and must always be at the beginning of the CD.

OptCD Overview

OptCD is separate program available to collect information that will allow ISOCD to optimize your disc layout. Every time your application reads from CD0:, be it a directory or a file, OptCD will note the access according to the type and level of the read. This data is used by Optimize in ISOCD to re-arrange the order of the directories and files. You can move the most often accessed files to the beginning of the CD or group together files that are often accessed after each other. The Optimize function of ISOCD takes into account the number of accesses when weighing which file or directory to move, so you should keep this in mind while walking your application through its paces.

A simple walkthrough, assuming you are simulating:

- Run OptCD with a statistics file.
- Walk the application on CD0: through its common usages.
- Turn OptCD OFF.
- Use Optimize in ISOCD to rearrange the entries with the statistics file produced.
- Rebuild image.

You may wish to work through this process several times to get the arrangement you are most happy with. We recommend keeping some of these layout files produced around until you are finished. In any case, you need to keep the final layout file!

You *must* have the layout file that produced the CD0: that you wish to optimize. OptCD records the reads of CD0: as blocks only, so determining which file or directory was read requires an exact knowledge of the source. If you created the image without any rearranging of entries and the source has not been changed, you can get away without a layout file the first time. Of course, once rearranged, only a layout file knows the order.

CDFS

OptCD normally monitors file reads and directory reads, whether they are caught in the cache or not. This gives equal access to all read activity. In this way, multiple iterations of the optimization process should produce similiar arrangements for the "most optimal." Cache hits are reads that do not actually read from the drive - but have been previously read into memory. Because part of the monitoring is actually within the file system, CDFS, multiple levels of monitoring were provided. Even though you probably are never going to need to know this information, we'll provide it anyway. You can monitor only actual drive reads, cache reads only, directory reads only, etc. There is even a monitor for all DOS packets if you have anything in particular you wish to parse yourself. With the Optimize function in ISOCD, you select which event type you wish to use, so monitoring anything extra does not hurt here.

CDFS MONITORS

0: All DOS Packets

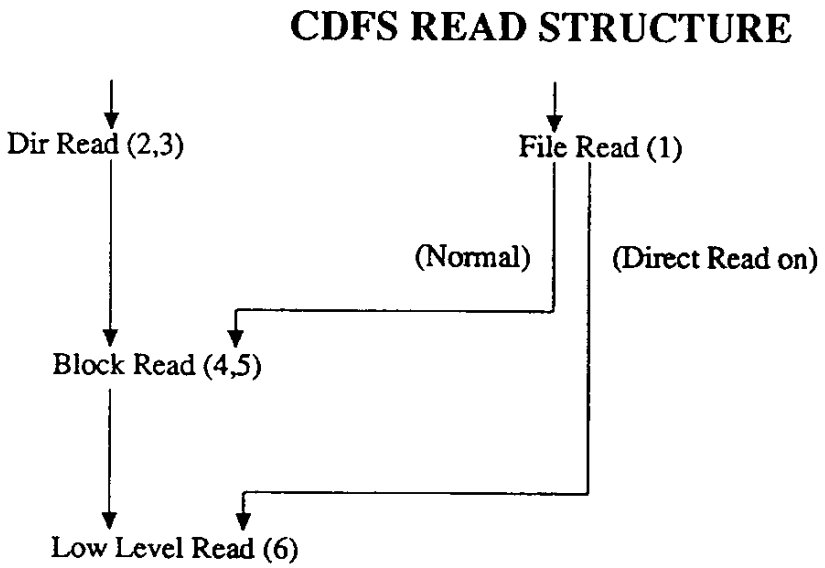
1: File Read

2: Dir Read (Not Cache)
3: Dir Read (Cache)

4: Block Read (Not Cache)
5: Block Read (Cache)

6: Lowest level read

The read monitors (1-6) are layered within CDFS as follows:



Briefly, both directory and file reads use the block system, except when Direct Read is on for a file or CD - refer to Direct Read under CDFS. In this case, file reads go directly to the lowest level. Directory reads implement their own caching system and the block level of the reads implement another. Blocks are 2048 byte sectors on the CD0: drive. If you wished to monitor only actual reads to the drive, you could monitor the lowest level only, level 6.

Statistics File Format

The statistics file is a text file with a line for each action. The first number on the line is the code, 0 though 6, for the type of action, e.g. 0=packet, 1=file read, etc. The second number, except for DOS packets, is the block read. DOS packet lines contain the actual three packet

arguments. This is mostly useless infomation since memory is dynamic, but its provided anyway. Both file read 1 and low level read 6, have a second argument, the length in blocks of the read.

An example listing, monitoring everything:

```
0 8 00000000 07cdc784 fffffffe
0 23 07ca0cac 07cd75f0 00f94dbc
2 32
4 32
6 32 9
0 15 07ca0cac 00000000 00000001
0 8 00000000 07cdc784 fffffffe
0 23 07ca0ccc 07cd75f0 00f94dbc
3 32
```

Batch

USAGE: OptCD [options] <Statistics file> [ON/OFF]

-b<n> buffer size in K (default 8)

-m<#...> events to monitor (default 123 - if not specified)

0: All DOS Packets
1: File Read
2: Dir Read (Not Cache)
3: Dir Read (Cache)
4: Block Read (Not Cache)
5: Block Read (Cache)
6: Lowest level read

[ON/OFF] Force OptCD on or off

Options:

<Statistics file>

The text file of activity from CDFS. If it exists, OptCD will just append more lines. Keep in mind that the size of this is related to the amount of activity, though it is still small.

-b<n>

This specifies the buffer size for messages from CDFS on the activity being monitored. You might want to enlarge this buffer if you have a lot of CDFS activity in a short period of time.

-m<#...>

These are the events that OptCD and CDFS can monitor. If you only wanted to monitor only directory reads, cached or not, you would specify:

OptCD <file> -m23

For an explanation of the events, refer to CDFS above.

[ON/OFF]

The first invocation of OptCD will turn it on and the next will turn it off. ON or OFF allows you to force OptCD on or off, this is useful in arexx or batch files.

General Errors

"Unable to open cdfs.library"

You have not started the simulation of CD0: or you have an older version of the CDFS part of the simulator.

"Cannot install notice"

OptCD could not create a message port named "MP_OptCD.Rendezvous".

"Cannot open Statistics file"

OptCD could not open the file.

"Not enough memory for messages"

There is not enough memory for the buffer. Don't forget to leave enough memory for the application you are going to run!

"Cannot get FSE port"

Is there another CDFS FSE monitor program running besides OptCD?

"Unrecognized message"

Not sure how this message could occur, it indicates a bogus message from CDFS.

"Failed to write statistics file"

Disk may be full.

Credits

Documentation by:
Kenneth Yeast

Software by:

Carl Sassenrath	[Alpha version of ISOCD, co-designer and producer]
Kenneth Yeast	[Graphical ISOCD, ability to rearrange files, and final version]

Special thanks:

Ben Phister	[helped at Commodore to shape ISOCD]
Chris Henry	[Hypermedia, provided invaluable beta testing with the Fred Fish CD-ROMs]

SimCD Overview

CDTV is more than just an Amiga: it is a low cost multimedia player, an Amiga married to a CD-ROM drive, a different beast all around. With CD-ROM as its medium, it brings to the user large amounts of data to make interactivity actually work. We are all just now seeing the real potential of this combination.

Developing for CDTV requires testing with all of this data in its final form, on a CD. This means expensive and slow mastering of test discs. SimCD changes the rules and allows you to create and test on the same development machine, cutting test discs only rarely. SimCD allows you to use a partition on a hard drive as the actual CD for testing. ISOCD is used to prepare this partition as a valid ISO-9660 image.

SimCD also uses the same ROM that is on CDTV, modified for simulation on the Amiga. Read, seek, and CDXL times are simulated, even considering the rotational latency of the CD. The infrared remote is simulated with the keyboard, bookmarks and cardmarks are available, preferences, screen saver, etc. are also simulated.

Because all of this is simulated on your development machine, you can also use enforcer, mungwall, io_torture, etc. This brings CDTV development to the same level as any other application, allowing you to deliver the same quality.

SimCD gives you a CDTV within your Amiga, allowing you to iteratively test your application without pressing a lot of CDs. Of course, you should still periodically cut a CD for testing on an actual CDTV. This gives you the final proof and the chance to test your application with a 68000, AmigaDos 1.3, actual memory left over from a boot, and the real CD-ROM drive.

Installation

We recommend using the new methodology of local directories in the standard system directories. In this way you can keep all of the non standard files separate from the standard AmigaDos files. For SimCD, create a "Local" directory in LIBS:, DEVS:, and L:; and add the following lines to your S:User-Startup file:

```
assign  LIBS:  LIBS:Local  add
assign  DEVS:  DEVS:Local  add
assign  L:     L:Local     add
```

The following files represent SimCD. Copy them to the appropriate directories.

File	Destination Directory
Devs/Local/bookmark.device	DEVS:Local
Devs/Local/cdtv.device	DEVS:Local
L/Local/CDFS	L:Local
Libs/Local/debox.library	LIBS:Local
Libs/Local/playerprefs.library	LIBS:Local
SimCD	Where programs normally go (BIN:, etc.)
BookFile	
FakIR	

Support for Simulator

SimCD consists of the following modules: SimCD, FakIR, BookFile, cdtv.device, bookmark.device, CDFS, playerprefs.library, and debox.library. Each can be used separately, though SimCD provides the primary control.

FakIR

FakIR simulates the CDTV Infrared Remote controller with the keyboard. In particular, the PLAY/PAUSE, STOP, FF, REW keys and the joypad are simulated.

FakIR can be run from within SimCD or from the command line. Running FakIR again or "FakIR OFF" will turn off the IR simulation. In addition, FakIR's operation may be suspended and re-enabled by pressing Ctrl, Shift, both Alt keys, and typing the toggle character (default: "i").

When FakIR is active, the arrow keys on the keyboard mimic the IR joypad. The PLAY/PAUSE, STOP, FF, and REW keys are mapped to the following function keys:

IR Key	Function Key
STOP	F1
REW	F2
PLAY/PAUSE	F3
FF	F4

When one of the above function keys is pressed, the rawkey code for the respective IR key is sent. The unusual behavior of the FF and REW keys is also simulated. The simulation aims for a general "feel" so that you can perfect your user interface.

Batch

USAGE: FakIR [options] [ON/OFF]
-t<c> toggle character (default 'i'), example -tc

To toggle the IR simulator on or off while running, hold down Ctrl, left Shift, both Alts, and the toggle character.

To remove the simulator, simply run this program again.

Options

[ON/OFF]
Force FakIR on or off. Useful in batch files and the like.

-t<c>
Change toggle character to <c>, useful in hand dexterity contests.

General Errors

"Cannot open console.device"
"Cannot install input handler"

These errors should never occur.

BookFile

SimCD allows you to load files into bookmark or cardmark memory and will create cardmark memory if not available. BookFile is provided so that you can retrieve information from a running CDTV as well as under the simulation. It provides more capability, allowing you to set up multiple test conditions, etc.

Batch

USAGE: BookFile [options] <file>
-l load memory from file
-s save memory to file
-c use cardmark instead of bookmark
-f create false cardmark memory for simulation
(implies -l (load) and -c (cardmark))

Options

<file>
File to be loaded or saved.

-l
Load file into bookmark or cardmark memory. This will copy the file exactly into the memory and thus must be the same size.

-s
This will save the memory to a file.

-c
Use cardmark memory instead of bookmark. If cardmark does not exist, BookFile will not be able to handle the request.

-f
Use for the initial creation of cardmark memory. The cardmark memory size is determined by the size of the file. -c and -l options are implied by this option. This option should only be used once, otherwise multiple cardmark devices are created.

Examples

Assuming you have created a particular set of bookmarks with your application on a CDTV and would like to use this set in you testing, use BookFile on CDTV as follows:

```
BookFile -s df0:Book.tst
```

Let us say that you a given set of cardmarks, such as special application data, and you wish to always start testing with this same set. Use BookFile the first time as follows:

```
BookFile -f Card.tst
```

And the following times as:

```
BookFile -c -l Card.tst
```

General Errors

"Cannot create StdIO"
BookFile could not create an IO request for bookmark.device or cardmark.device.

"Could not create false cardmark memory"
BookFile was not able to create cardmark memory, probably not enough memory, the file was too large, or the file was not the right file.

"Cannot open device"
BookFile could not open bookmark.device or cardmark.device.

"Not enough memory"
BookFile needs 32K for buffers to handle the bookmark or cardmark devices.

"Cannot open file"
BookFile could not open the bookmark or cardmark files.

"I/O error sizing memory"
Bookmark.device or cardmark.device may be corrupt, they were not reporting sizes correctly.

"File size does not match memory size"
BookFile currently only supports dump files that are exact copies of the bookmark or cardmark memory. If the file is a different size, then they do not match.

"Cannot reformat memory"
"Cannot load memory"
The bookmark or cardmark device could not handle the information. They might be corrupt. Remember that both devices and their information are in the regular memory area and can be affected by any application running. Make sure you use enforcer and mungwall when testing your applications.

"Cannot dump memory to file"
BookFile could not get the memory from the bookmark or cardmark. They might be corrupt, see above.

"Cannot write memory to file"
BookFile could not write to the file.

DEVS:cdtv.device

Cdtv.device is basically the same device driver used on CDTV, except for being AmigaDOS loadable, the ability to use scsi.device and thus a hard drive partition, and seek timing simulation. It is used when SimCD sets up the simulation of CD0: with CDFS, but it can also be used separately for other partitions by creating your own mountlist entry. It is assumed that in doing this you are fully aware of how to set up a mountlist entry. If you are not familiar with this, please get some help - this is the quickest way to grunch your entire system.

In any case, your mountlist must contain CDFS as the file system, since it is needed to interpret the ISO-9660 image that ISOCD will place there. An example of one follows:

```
CDX:
    Device          = cdtv.device
    Filesystem      = L:CDFS
    Unit            = 6
    Flags           = 0
    Surfaces        = 1
    BlocksPerTrack  = 553
    Interleave      = 0
    Reserved        = 2
    LowCyl          = 2
     HighCyl        = 637
    Buffers         = 2
    BufMemType      = 1
    Mount           = 1
    GlobVec         = -1
    StackSize       = 4096
    DosType         = 0x43443031
#
```

Once loaded into memory, cdtv.device cannot be expunged. It must be placed in the DEVS: directory, preferably in the DEVS:Local directory. Refer to the CDTV documentation on cdtv.device for information on dealing with it directly.

Commands

These are the commands documented in the cdtv.device autodocs and notation on what is simulated.

Command	Simulation	Notes
CDTV_ADDCHANGEINT	Not Useful	
CDTV_CHANGENUM	Not Useful	
CDTV_CHANGESTATE	Not Useful	
CDTV_FADE	Simulated	No actual audio
CDTV_FRAMECALL	Simulated	75Hz timing!
CDTV_FRAMECOUNT	Simulated	
CDTV_FRONTPANEL	Ignored	
CDTV_GENLOCK	Ignored	
CDTV_INFO	Ignored	
CDTV_ISROM	Simulated	Needs the TOC file
CDTV_MOTOR	Not Useful	
CDTV_MUTE	Simulated	No actual audio
CDTV_PAUSE	Simulated	
CDTV_PLAYLSN	Simulated	No actual audio, needs TOC
CDTV_PLAYMSF	Simulated	No actual audio, needs TOC
CDTV_PLAYSEGSLSN	Simulated	No actual audio, needs TOC
CDTV_PLAYSEGSMSF	Simulated	No actual audio, needs TOC
CDTV_PLAYTRACK	Simulated	No actual audio, needs TOC
CDTV_POKEPLAYLSN	Simulated	No actual audio, needs TOC
CDTV_POKEPLAYMSF	Simulated	No actual audio, needs TOC
CDTV_POKESEGLSN	Simulated	No actual audio, needs TOC
CDTV_POKESEGMSF	Simulated	No actual audio, needs TOC
CDTV_QUICKSTATUS	Simulated	
CDTV_READ	Simulated	
CDTV_READXL	Simulated	
CDTV_REMCHANGEINT	Not Useful	
CDTV_RESET	Ignored	
CDTV_SEEK	Simulated	
CDTV_STOPPLAY	Simulated	
CDTV_SUBQLSN	Not Useful	
CDTV_SUBQMSF	Not Useful	
CDTV_TOCLSN	Simulated	No actual audio, needs TOC
CDTV_TOCMSF	Simulated	No actual audio, needs TOC
CDTV_WRITE	Fully Simulated	Not!

DEVS:bookmark.device

Bookmark.device controls bookmark and cardmark memory. Make sure that it is in the DEVS: directory, preferably in the DEVS:Local directory. Like cdtv.device, it is not expungable, since the memory it deals with is supposed to hang around. Refer to BookFile above and the CDTV documentation on direct programming of the devices.

L:CDFS

CDFS is the file system that allows you to use an ISO-9660 image as a normal AmigaDos volume. It is set up automatically for you through SimCD to simulate CD0:. It must be in the L: directory, like the others, preferably in the L:Local directory. When you use it in your own mountlists, you can use cdtv.device or scsi.device as the device for the partition. If you were to use scsi.device, CDFS would use the partition at full speed, without seek timing simulation or any cdtv.device code. This is useful when you wish to use an ISO-9660 partition for other than simulation. Since this format is read only, there is no need for any of the dynamic file handling. Files are always in one piece, no fragmentation or scattered layout of the files.

To use independent of SimCD, create a mountlist entry for the partition that will hold the image or use HDToolBox to set up a partition. If you use HDToolBox, follow these steps:

- Make sure that SimCD is installed on the system, since HDToolBox will look for CDFS.
- Start up HDToolBox.
- Select the drive where you want the ISO-9660 partition.
- Click on "Partition Drive", this will take you to the Partitioning control panel.
- Select or create the partition to contain the image. Be sure to make it large enough, at least as large as all of your files and try to have room for the final application size.

  You can use an entire drive for the simulation or just a part of it, sharing the drive with the source material, for example.

  Do *not* chose "CD0:" as the name, SimCD uses that as the simulated drive name.

- Click on "Advanced Options" to gain access to the file systems settings.
- Click on "Add/Update File Systems", this will take you to the File System Maintenance control panel.

- Click on "Add New File System".
  - In the requester that appears, enter the following:
  - In the Filename, enter "L:CDFS."
  - In the DosType, enter "0x43443031."
  - In the Version, enter "25."
  - Click on "OK."
- The custom file system should appear in the list, click on "OK."
- Click on "Change File System for Partition," this will take you to the File System Characteristics panel.
  - Click on "Custom File System," this will also enable the "Identifier" string gadget.
  - Change the text in the "Identifier" string gadget to "0x43443031."
  - Click on "OK."
  - Click on "OK" again to return to the main control panel.
- Click on "Save Changes to Drive." This will write the new RigidDiskBlock out to the drive with the CDFS file system on it. Be sure to do this before rebooting!
- Click on "Exit," HDToolBox should ask you to reboot, do so now.

The CDFS partition should now be available to AmigaDos. Now it is ready for ISOCD to write the ISO-9660 image to it. If a new version of CDFS is ever released, you will need to use HDToolBox again to update the custom file system.

Until ISOCD places a valid ISO-9660 image on the partition, you will notice long pauses at boot time. This is because CDFS is trying to find a valid PVD, or Primary Volume Descriptor, on the image. It will read 256 sectors in its attempt to find a valid PVD. This will cause a long pause, as if the drive was being validated, but you do not need to worry. It will never be seen again, once you have an ISO-9660 image on the partition.

> *Note:* This version of CDFS retains the date error present in CDTV V1.00 that does not report ISO-9660 dates properly. It will show a date one day later then is correct in most cases.

LIBS:playerprefs.library and LIBS:debox.library

Playerprefs.library and debox.library are provided and should be placed in LIBS:, preferably the LIBS:Local directory. Refer to the CDTV documentation on the usage of these two libraries. Bookit is usable to set the preferences usage, refer to the bookit documentation. Keep in mind that playerprefs.library is 170K and if you invoke the bouncing logo screen blanker, even more memory is needed.

Memory/Resource Usage

SimCD itself uses little resources, but does provide a guess as to the probable memory usage of the options chosen to be simulated. Also, SimCD requires AmigaDos 2.0 or higher.

Once Simulator is Running

Once the simulator is running and CD0: is available, you can still run SimCD to change some of the options. CDFS and the three device drivers, cdtv, bookmark, and cardmark, are not removable once you have started them. If CDFS/cdtv.device is chosen, Sim Device and TOC files cannot be changed. If bookmark or cardmark is chosen, the corresponding files cannot be changed.

Files

Settings

If no settings file is chosen, SimCD will use S:SimCD.Default. Load and save options are available from the menus. Use multiple files for separate applications or test configurations.

Sim Device

The Sim Device specifies the hard disk partition that is to be used for CD0:. It must have been created with either a mountlist entry or through HDToolBox, see CDFS under Settings for information on creating a partition. The partition can be one that is already controlled by CDFS without cdtv.device. Remember to not enter CD0:, since CD0: will be the drive that results from the simulation. The Sim Device is the same one that ISOCD uses to write to as the Image drive. This option is unchangeable once used.

Book Mark

Book Mark refers to the file loaded into the bookmark memory. You can use the BookFile program to capture bookmark memory from CDTV, refer to the BookFile section under Support for Simulator. Like Sim Device, this option is unchangeable once used.

Card Mark

Just like Book Mark, this specifies the file for cardmark memory. If there is no cardmark memory, it will be created. The BookFile program also handles cardmark memory. Card Mark is also unchangeable once used.

TOC

The TOC file specifies the CD Table of Contents for cdtv.device to simulate. It specifies the types and sizes of the various tracks to be simulated. The sizes are in frames (sectors). Specify Data or Audio for the type, a tab or spaces, then the size.

ISOCD will provide you with the size of the data track or ISO-9660 image.

For example, if the ISO-9660 image used 64 megabytes and was followed by three audio tracks, the TOC file would contain:

```
Data  32768
Audio 15750
Audio 45322
Audio 4500
```

Cdtv.device uses this information for replying to the TOC commands. This file is also unchangeable once CD0: is simulated.

Settings

These are the settings for the simulation. CDFS and cdtv.device are necessary for simulation of CD0:. A lot of these options will remain on after simulation until a reboot. The two libraries can be purged as long as they are not currently opened; IR and No Fast Memory are always available to be turned on or off.

CDFS

CDFS is necessary for simulation of CD0:. You need to set the Sim Device to the partition that you have set aside as the ISO-9660 volume. You also need to create this partition first with ISOCD before simulating it. Cdtv.device is necessary if this option is used. After simulation you can find the file MountCDTV.tmp in the T: directory. It is the mountlist entry that mounted CD0: if you wish to see how it was mounted.

cdtv.device

Cdtv.device will use the TOC file if you specify it, but it is not necessary. The TOC file would be needed if you have any audio tracks used in your application. Refer to TOC above.

bookmark.device

The bookmark.device will load the Book Mark file if specified, otherwise it will setup the bookmark memory area blank. This device is needed if you are going to use cardmark.device, since bookmark.device is used to create it. BookFile is available for further management of bookmark.device.

cardmark.device

The cardmark.device requires the Card Mark file and bookmark.device set. The Card Mark file will be loaded into card mark memory. This memory is not recoverable.

playerprefs.library

The playerprefs.library handles CDTV preferences and audio control. Of course, the audio panel is not usable, but you should use the preferences.

debox.library

Debox.library is available for the decompression routines. Both playerprefs and debox are documented in the CDTV documentation.

IR

IR turns the Infared Remote simulation on or off. Refer to FakIR under Support for Simulator.

No Fast Memory

No Fast Memory just calls NoFastMem. Since CDTV has no fast memory, this option is placed here for convenience.

Speed

Speed is available for future use.

Avail

Avail reports an approximation of available memory after choosing Simulate.

Simulate

Simulate starts the requested devices, loads the appropriate memories, etc. After everything is started, SimCD will save the current settings to the Settings file and exit to the shell. If you do not want to simulate, simply close the window, chose Quit from the menu, or hit Amiga-Q.

Load/Save Settings

Both load and save are available from the menus for the settings file. Use these for creating an alternate settings file for a specific application. Then use SimCD for future simulations as:

SimCD -b <file>

Batch

USAGE: SimCD [options] [<Configuration file>]
-b batch execute (do not use interface) (requires Configuration file)

Options

<Configuration file>
This is the file that contains the SimCD settings. It is normally S:SimCD.Default if not specified on the command line.

-b
This will configure the Simulator based on the configuration file. If one is not specified, it will use S:SimCD.Default. This the common usage of SimCD once the settings are chosen for normal usage.

General Errors

"Cannot open settings file"
Configuration file is not available or is corrupted. Remove S:SimCD.Default, run SimCD fresh, and reset all of your settings.

"Intuition (V2.0) not available"
SimCD requires v2.0 of intuition or greater. If you specify -b, SimCD will not use the interface and will not require v2.0.

"Interface not available"
Probably not enough memory for window.

"Device <dev:> not available"

Dev: specified in Sim Device was not found. Check that it is properly set up with HdToolBox or that your mountlist entry is correct. See CDFS under Settings.

"TOC file <file> not available"

SimCD could not open the TOC file.

"More than 99 entries in TOC file"

Just like a CD, SimCD can only handle up to 99 tracks, data or audio. This error message may also occur if you attempt to load an invalid TOC file, the TOC file interpreter is fairly simple.

"TOC file error, line #n"

Text in TOC file was not of the form "Data 32768" or "Audio 45322" in line n. Make sure that you are preparing the TOC file with a text or word processor that saves out text files. Do not use just any file.

"Cannot create StdIO"

This error should not occur.

"Cannot open device"

SimCD could not open bookmark.device or cardmark.device. Check that SimCD has been properly installed and these devices are in DEVS: or DEVS:Local. Check Installation.

"Not enough memory"

SimCD needs 32K for buffers to handle the bookmark or cardmark devices.

"Cannot open file"

SimCD could not open the bookmark or cardmark files.

"I/O error sizing memory"

Bookmark.device or cardmark.device may be corrupt, they were not reporting their sizes correctly.

"File size does not match memory size"

SimCD currently supports only dump files that are exact copies of the bookmark or cardmark memory. If the file is a different size, then they do not match.

"Cannot reformat memory"
"Cannot load memory"

The bookmark or cardmark device could not handle the information. They might be corrupt. Remember that both devices and their information are in the regular memory area and can be affected by any application running.

Glossary

CD0:

The CD-ROM drive on a CDTV or the partition simulated as this drive with SimCD.

CDFS

The CDTV CD File System. This allows you to treat CD0: as just another Amiga drive, even though it is in ISO-9660 format.

cdtv.device

The device that controls the CDTV hardware or the simulation of it with SimCD.

CDXL

The fastest way to transfer data from the CD-ROM drive into the CDTV. CDXL basically starts a read and begins transferring the incoming data into places in memory that you specify. CDXL nodes control the shotgun effect of the data being read in, since the read does not pause. CDXL removes the liability of the CD-ROM seek times, except for the initial seek.

ISO-9660

Refers to the 1988 ISO-9660 standard, specifically ISO 9660: 1988 (E), corrected 1988-09-01. This is the standard produced by ISOCD.

Credits

Documentation by:
Kenneth Yeast
Carl Sassenrath [Previous documentation]

Software by:
Kenneth Yeast
Carl Sassenrath [Author of ROM software, Producer, and "Father" of CDTV]

Also a special thanks to Leo L. Schwab, and everyone at Silent Software for their work on CDTV.

◆