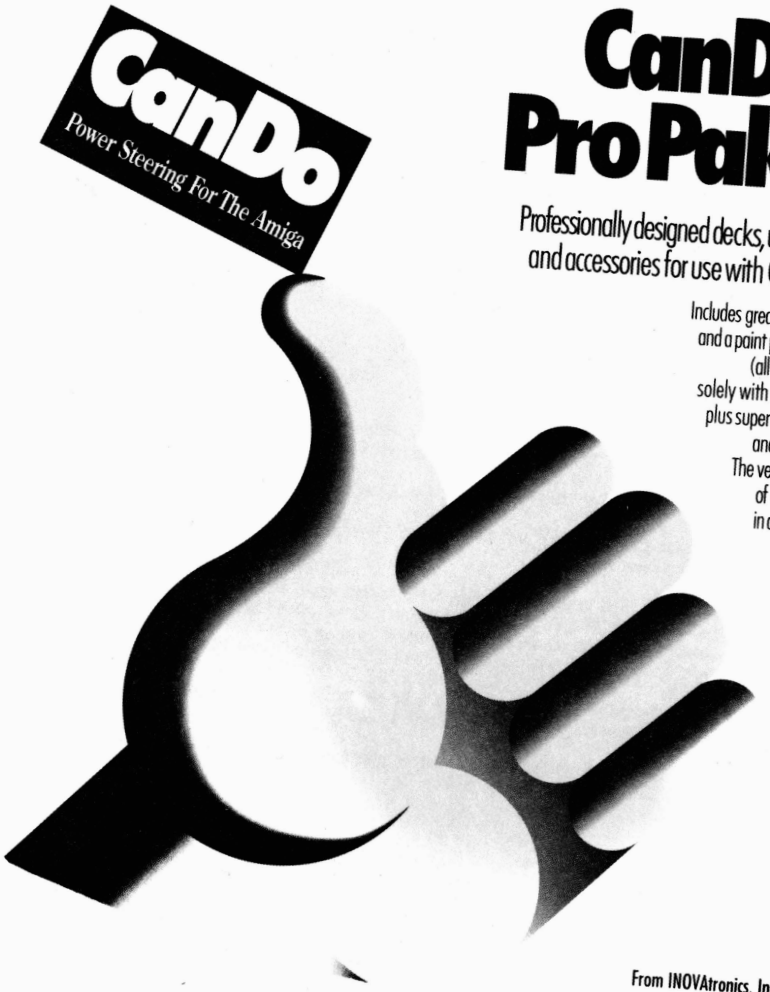

CanDo Pro Pak 1

Professionally designed decks, utilities and accessories for use with CanDo.

Includes great games and a paint program (all created solely with CanDo), plus super utilities and more. The very best of CanDo in action!

From INOVAtronics, Inc.
8499 Greenville Avenue Suite 209B
Dallas, Texas 75231 1-214-340-4991

Welcome to **Pro Pak 1:The Best of CanDo**

This collection of CanDo decks, utilities, and other resources was designed to serve a number of purposes for CanDo users.

The decks were designed to show the many different things that can be done with CanDo. Each of the decks is documented to show how the executable version works, as well as insights on how they were created.

The KeyInput object, the Layout editor tool and the Cross-Referencer were designed and included in this package to increase the power and facility of CanDo. The Layout tool and, especially, the KeyInput Object should be of interest and great use to all CanDo users, while the cross-referencer is aimed at slightly more advanced users.

The IFF resources were made freely redistributable in the hope that you, the CanDo user, will find them useful when creating your own decks. In fact, that is the aim of everything in this package. Enjoy!

Pro Pak 1 Legalities

Pro Pak 1
Table of Contents

"ShootingGallery"

How To Play ShootingGallery

Step right up, everyone's a winner in the CanDo ShootingGallery! Step right up! Release your aggressive, hostile, or even downright violent feelings by blasting ducks, bears, and spinning lolly-pops, as well as six annoying targets.

Shoot as long as you like; don't worry we'll make more. In fact, this game goes on and on. You can run out of ammo however, so be prepared to ante up by buying some more shots from the vending machine.

As in any shooting gallery, the greater accuracy yields greater rewards, so hit the target in the bullseye when you can.

To quit, just select "Quit" from the menu or press the quit button in the upper-left corner of the window. Have fun!

How ShootingGallery Works

ShootingGallery is a single Card Deck.

As with other games designed with CanDo, the colors on the display are used extensively to determine on what the player has clicked. Within the context of its display area, each BrushAnim has a small set of possible colors which constitute a hit. In the case of the ducks, the rabbits, and the spinner, two colors, 8 and 9 are shared by the targets. Processing is simple; all clicks on colors that are not any of the target's colors are not a hit.

The most complicated aspect of ShootingGallery is the design of the BrushAnims it uses. The animations used in this Deck were carefully designed to allow PingPong animation to handle as much of the movement and decision making process as possible.

The ducks move in a continuous stream from left-to-right. Once they are started in the script, they run without intervention. A single AreaButton surrounding them ("DuckArea") handles all mouse clicks in their display area.

The rabbits and the spinner both use a simple **AtDestination** script to detect when the BrushAnim has reached the point at which it must be turned. The spinner was also designed in PingPong mode with DeluxePaint III, so that the up and down motion of the animation is entirely processed by CanDo by application of the PINGPONG flag after the animation was loaded. This reduces greatly the work required to keep these animations running.

The target BrushAnims also work in PINGPONG mode. That keeps them moving up and down continuously. They are comprised of three different colors, allowing a higher score to be awarded for hitting closer to the center of the target.

Each target BrushAnim also has a Reoccurring Timer associated with it. The purpose of these timers is to vary the pattern of the targets as they move up and down on the display. Each Timer is set for a slightly different rate, and each checks to see if its BrushAnim is currently visible or not. If it is not visible and it is in motion, it is deactivated until its next timer event. Without these timers, the target BrushAnims would all move up and down together, resulting in a very dull display; with them, the targets change their patterns in an unpredictable way.

The Routines in ShootingGallery are as follows:

CheckForBearHit

Each time one of the bear Buttons (1-4) are hit, this Routine decides whether or not a bear was actually hit (by examining the color the mouse pointer was on). If a bear was hit, the bear says "ow!", otherwise the "MissedShot" sound is played. Hitting a bear does not affect the score.

CheckForHit

This one Routine handles hits in the rabbit, ducks and the spinner areas. It accepts three arguments. The first is the sound buffer name to play if a good hit was made. The next argument is the number of points to add to the player's score on a hit. The third argument is the name of a handler for this target. For the rabbit, the handler is the "TurnRabbitAround" Routine, while for the spinner it is the "TurnSpinnerAround" Routine. The ducks require no handler, as they do not change behavior when hit, so no handler is specified. The Routine checks to be sure that a HANDLER was passed to it before **Do**ing the HANDLER.

CheckForTargetHit

When one of the bullseye targets is hit, this Routine checks to see which part of the target got hit. The outer part of the target (color 10) is worth 10 points. The next section (color 12) is worth 25 points, while the bullseye itself (color 6) earns 100 points. If none of the parts of the target were hit, the "MissedShot" sound gets played. By using this Routine which determines by color whether or not a target was hit, a rectangular AreaButton can control an irregularly shaped target.

InitAllResources

By pre-loading as many of the resources used by this Deck as possible, each resource can be given a shorter name, hence every file that is loaded here is given a different buffer name. Additionally, the BrushAnim flags that must be set for each animation are set here. If the BrushAnims were not pre-loaded, there would be no opportunity for setting these important flags. Note the use of

DECOMPRESSED mode; this allows the BrushAnims to run at the greatest speed possible. Also, the use of PINGPONG mode with most of the BrushAnims relieves the logic of the program from the responsibility of dealing with the animations in many circumstances.

ShowScore This Routine is in charge of updating the SCORE variable, as well as showing its current value. It can also be programmed to change the score to 0 by passing a value of -1 to it.

TurnRabbitAround Whenever either of the rabbit animations reach the edges of their movement areas, this Routine removes the rabbit BrushAnim, and shows the animation which will move the rabbit in the opposite direction. This Routine will also be called if the rabbit is hit, thus changing its direction. Note the use of the RABBITGOINGRIGHT variable to keep track of which way the rabbit is current moving, and thus which animation is playing.

TurnSpinnerAround The spinner is a simpler matter because only one animation is involved. When the spinner reaches its destination, or is hit by the player, the animation is simply made to move in the opposite direction.

UpdateShotCount Each time the player clicks in the window, the number of shots he or she has remaining is reduced by one. There are buttons on the display that will "buy" more shots.

"Solitaire"

How To Play Solitaire

This is the game of Solitaire, implemented using Klondike rules. Use the mouse to click on cards in order to move them around. The "Rule of 3" is used to get cards from the deck at the top; click on the face-down card on the right to deal the next 3 cards (top card shows on the left). Click on the left stack to pick up the card. There are menu options to select 1 card per deal or 3 cards per deal. Three cards per deal is the default.

The "Rule of Kings" is not implemented. This means you can place any card or stack on top of any empty stack or face-down card, as well as on any legitimate face-up card. By legitimate face-up card, we mean alternating-suit, next-lower-value cards, by themselves or at the top of the picked-up stack, can be placed on any bottom card showing on a stack in the row of seven stacks, and only same-suit next-higher-value cards can be placed in the discard stacks on the left. Any Ace can be placed in any open position to start.

To pick up a face-up card, just click on it. If it is a face-down card in a stack on the row of seven stacks, clicking on it will turn it face-up.

To pick up a stack, click on the exposed part of the top part of the stack that you want. For example, if you have a K-Q-J-10 that you want to move to an open spot, click on the portion belonging to the King. The pointer will become the whole stack. Just position the stack over the destination (anywhere over the destination stack should work) and click to release. Note: Of course, if the bottom card of the destination stack is not a different-color, next-higher-value card, the placement will not be allowed.

As with a real deck of cards, there are ways to cheat... but that's between you and your conscience.

To quit, use the menu option, or press Right-Amiga-Q.

How Solitaire Works

The Solitaire game has two cards, routines to handle the drawing and analyzing of cards, and several AreaButtons on the main screen, one for each of the seven stacks, the four ace stacks, and the deck areas. As you will find with most of the CanDo Decks in this package, Routines are used heavily to reduce the amount of **If-Else-EndIf** processing that must be performed, as well as to make it easier to later change the logic used in the Deck.

Each stack of playing cards is controlled internally by a CanDo Document. These documents have several lines of text stored in them, one of each playing card on the stack at any given time. These Documents are never displayed, however, because they are not linked to either a Memo Document Object or a List Document Object. They are used solely for the purpose of storing and manipulating the playing cards. Note that CanDo programmers often use Documents to replicate the functioning of a "stack" or an "array". These are data storage constructs found in other languages. CanDo has

no arrays or stack, but the logical cursor inside each Document can be positioned and maneuvered to simulate these constructs.

Another Document, called "StkLevel", contains a count of the number of playing cards in each of the seven stacks. The **PositionOnLine** command is used to move the Document's internal cursor to the line number containing the number of playing cards in a given stack. For example, in the Routine "PlaceCard", the variable Showing is set to the number of cards in the current stack by the use of the following script lines:

```
WorkWithDocument "StkLevel"
PositionOnLine StackNumber
Let Showing = Integer(TheLine)
```

In this case, the variable StackNumber is the number (1-7) of the current stack of playing cards being examined.

The Cards are:

GenPicCard	This Card creates a picture of the Solitaire screen,for fast redraw of screen between games. Afterwards, a TimerObject Event Script moves us to the SolitaireCard card.
SolitaireCard	The main Card, it creates the playing card brushes (through ClipBrush), and runs the game.

The Routines are:

ACrd	This Routine draws the playing card designated by the current active stack to the given coordinates. If there is not a card to display, it displays an empty pile marker.
AGetValue	This Routine converts from single letter playing card names, like "2" or "J", into the values 1-13 which will be used to reference a given playing card.
ClickOnStack	Whenever one of the AceStacks or the CardDeck is clicked on by the player, this Routine is given the name of the Document which has the information about what's in that stack. If you are holding a card, it will place the card on the stack, otherwise it will try to pick up a card, if one exists, from the stack.
GetCard	This Routine removes a single card off of a stack of cards, clipping the image of the card so it can be moved around, and draws the card which was beneath it.

PlaceCard	Places a card onto a stack of cards, making sure that it is within the rules place the card on the stack.
PlaceStack	This does the same thing as PlaceCard, but it is used when a stack of cards is being moved rather than just one.
WorkWithStack	Each time one of the seven CardStacks is clicked, this Routine is invoked, and is passed the stack number (1-7) as an argument (ARG1). It looks at the MouseY to determine which playing card in that stack the user clicked. If the MouseY is below any possible card, the Routine just exits. Otherwise, if the user currently has a card or group of cards picked up, the appropriate Routine is called to put them down, if it is legal to do so. If none of the above rules apply, then if there are cards showing on that stack, they get picked up by a ClipBrush command and the pointer gets set to the clipped image, else if the top card on the stack is face down, it gets turned over.
ZipCard	This Routine is responsible for creating the card imagery used in this game. It first draws the images onto the screen, then clips them as brushes for later use. The images it creates include the arrow pointer and the images for the spade, club, heart, and diamond cards.
ZipCardFrame	Draws the frame of a card. This routine is used by the routines which create the imagery needed by this game. Once the imagery is created and clipped as brushes, this routine is no longer needed.

"CanDoMan"

How To Play CanDoMan

If you were a happy, smiling Bot like CanDoMan, what would you do if you were being ruthlessly pursued by BadDiskBots? Run away, that's right! That is one aspect of what CanDoMan is all about. Along the way, you can also pick up Treasure, Keys to open doors to other levels, and Magic giving you the strength to temporarily kill your enemies.

To start CanDoMan, run it from the Workbench or from the CLI using DeckRunner or DeckBrowser. Once it loads, the starting horn will blow and you're ready for a new game. Select

"New Game"

from the menu and CanDoMan will appear in the maze followed by the evil Bots. As each Bot enters the maze, it will announce itself - Brenda Bridgeboard, Sally Silicon, Billy deBot, and Fred.

Use the arrow keys to move CanDoMan around the maze. Each red dot in your path is a Treasure, while the Magic (diamond shaped) and the Key have higher values. Once you have picked up some Magic, you can temporarily kill Bots for the next several seconds. If you run into a Bot without this power, you are likely to lose a life, and you only have four lives to lose.

The Key allows you to move through locked doors to other levels of the maze. CanDoMan comes with four levels already, but you can easily make your own additional levels.

At certain points in the maze, WormHoles (shaped like concentric squares) will move you to a new random position on the screen. This is a great way to get away from a pursuing Bot, but you never know where you'll end up. Level 1 has only a few WormHoles, while Level 4 is essentially *composed* of them.

You can pause the game, end the game, start a new game, or quit CanDoMan, by selecting the appropriate choice from the menu. Each menu selection also has a right-Amiga key equivalent, so you can perform those actions from the keyboard also.

How CanDoMan Works

CanDoMan consists of one Card with a PictureWindow in it. The PictureWindow displays an image called "BackGround.Ilbm" from the "CanDoMan/images" directory. This picture is lo-res with a thick border along the edges and an information display area at the top. Inside these bounds, each maze is displayed using the ShowBrush command.

The mazes of CanDoMan were designed using DeluxePaint III from Electronic Arts. Each maze is a low-res brush image and is a grid comprised of 19 squares horizontally

by 10 squares vertically. The brushes are stored in the "CanDoMan/images" directory under the names "Level.1" through "Level.4". Every element of the game has a distinct color combination. By examining these colors with the ColorOfPixel function, the characteristics of each square can be determined. A worksheet showing all of the game elements is provided.

CanDoMan design is broken loosely into the following aspects:

Analysis of the nature of a given location in the maze
Resolution of conflicts between CanDoMan and the things in the maze
The timer that makes the Bots move and controls the duration of certain attributes
Handling of the arrow keys

At the heart of the game is the routine **QueryLocation**. This is used to determine whether a particular square of the maze can be moved into, and, if so, if it is currently occupied by another Bot or some other game element. Two variables, BLOCKED and EMPTY, are set to YES or NO as appropriate by this routine.

QueryLocation accepts two arguments, an X and a Y location. It checks first to be sure that the location is within the bounds of the 19 by 10 game area. If it is not, then the location is assumed to be BLOCKED. If it is within bounds, the ColorOfPixel function is used to get the color of the pixel in the center of the square. (A formula is used to compute the location of the center of the square, accounting for the square X,Y location and the width and height of the borders of the main game area.) The brushes used for the various game elements are specifically designed to have a unique color at their center pixel. In this way, **QueryLocation** can precisely identify which game element is at any location by sampling its color.

If color 5 is in the square, then the location is BLOCKED, because 5 is the color used for the green maze walls. If it is not color 5, then the square is not BLOCKED. However, it might still have some other game element in it, so a comparison is made to the empty square color (color 0). The EMPTY variable is then set to NO if any other color is present.

The next routine to examine is **QueryInteraction**. This routine will decide what action to take for each type of game element in a square when either CanDoMan or a Bot tries to move into that square. The possible game elements, and their respective colors, follow.

Empty	0	Treasure	2
Wall	5	Bot4	10
Magic	11	Bot1	16
Door	18	LockedDoor	19
Bot3	22	WormHole	25
Bot2	28	Door	29
CanDoMan	30	Key	31

There is a Routine to handle each color. Each one is named **Interaction** with the color number added to the end of the name, as in, **Interaction_0** or **Interaction_10**.

When a Bot is being moved, **QueryInteraction** is called with the new X and Y locations passed as arguments. When CanDoMan is moving, a third parameter is also passed which is a Boolean TRUE. This tells **QueryInteraction** that CanDoMan is moving, and not a Bot. This fact is used by each of the **Interaction** Routines in order to determine whether a move is legal or not, or has special consequences, based on the identity of the moving piece. In other words, if a Bot runs into another Bot, a certain sound is played, whereas if CanDoMan runs into a Bot, he might kill the Bot or get killed by the Bot.

Each **Interaction** Routine has the option of declaring a HANDLER Routine for this type of interaction. When CanDoMan moves into a square occupied by treasure, the Routine **PlayerVsTreasure** is declared to be the HANDLER to resolve this conflict. The HANDLERS that exist are:

BotVsBot	BotVsDoor
BotVsKey	BotVsLockedDoor
BotVsPlayer	BotVsTreasure
BotVsWormHole	PlayerVsBot
PlayerVsDoor	PlayerVsKey
PlayerVsLockedDoor	PlayerVsPlayer
PlayerVsTreasure	PlayerVsWormHole

Note that the HANDLER is not invoked by **QueryInteraction**, nor by its **Interaction** Routine. The HANDLER is called by the Routines that make the Bots move at regular intervals, and by the Routines handling the keyboard input which moves CanDoMan.

For each Bot, there is a set of Routines for dealing with various aspects of their render and movement. The following is an explanation of the Routines that apply to Bot1. They also apply to each of the other Bots, with the appropriate substitution of name and number.

StartUp_Bot1

This Routine is called to start Bot1 in the maze. When the game is over, or before it starts, the variable LEVEL is set to 0, so this Routine checks to see whether or not the Bot should be started at all. If it is started, the Bot is placed on the display via the **PlaceBot** Routine. Then the variable BOT1_REVIVE is set to FALSE, indicating that the Bot does not need to be revived, because it is already in the maze.

ShutDown_Bot1

When CanDoMan moves from one level of the maze to another, each Bot is shutdown to stop it from moving. This also happens when the game is stopped.

Place_Bot1

This Routine finds an acceptable location to place Bot1 in the maze. It runs in a loop, looking for an empty spot, or a spot which contains treasure. (A Bot can move

into a square with treasure and move out of it again, leaving the treasure behind.) When a good spot is found, the **Show_Bot1** Routine is called to update the display.

Move_Bot1

A single Reoccurring Interval Timer ticks 12 times per second. This timer makes the Bots move. This Routine is called each time the timer's occurred script happens. If the Bot is dead, nothing will happen. If the Bot should be revived, (it has been temporarily killed by CanDoMan) it will be revived and placed in the maze via **Place_Bot1**. The next test determines if the Bot will randomly not move at all. Finally, if none of these other conditions apply, the Bot will be moved either vertically or horizontally (decided randomly) if and only if it can move to its new location legally. The actual movement is accomplished by the **FinishMove_Bot1** Routine.

FinishMove_Bot1

This Routine moves the Bot to its new location. If the square the Bot is moving into is empty, the old square is cleared, the new coordinates saved, and the Bot shown in the new spot. If the square is occupied by something, the **QueryInteraction** Routine is called to determine which HANDLER must be used to resolve the conflict. The HANDLER is then called, and it is responsible for changing the display appropriately.

Kill_Bot1

Whenever Bot1 is killed, this Routine plays a sound, and scrolls the Bot into the floor of the maze. It sets the BOT1_REVIVE flag to TRUE so the Bot can come back later. If the Bot was carrying the Key, the Key is "dropped" by resetting the BOT1_KEY flag and showing the key to be where the Bot was.

Erase_Bot1

When the Bot was originally placed in the square from which it is being erased, anything else that was in the square was

clipped into a Brush Buffer called Bot1Clip. When the Bot is erased, the clip is shown to replace what the Bot was covering. The clip can be empty, so if it is, the square is cleared with a simple **AreaRectangle** command.

Show_Bot1

To place Bot1 visually on the display, the current contents of the square into which Bot1 is moving are clipped into the Brush Buffer Bot1Clip. Then the image of the Bot can be safely stamped into the square with a **ShowBrush** command.

KeyFlag_Bot1

This Routine handles the special case when Bot1 has picked up the Key. A TRUE or FALSE status is passed as an argument and the BOT1_KEY Routine is set accordingly to this value. The Key is a special case because it the only type of treasure which a Bot or CanDoMan can pick up and move to another spot.

There is a similar set of Routines that handles CanDoMan. They are:

Place_Player

This Routine will randomly place CanDoMan in the maze using similar logic to that used to place the Bots.

Move_Player

The four arrow keys are each monitored by an AKey Object. When they are pressed, the AKey Objects call this Routine and pass to it offset from the current location of CanDoMan, which reflect what direction and how far CanDoMan is to move. The X offset is given first, followed by the Y offset, becoming ARG1 and ARG2 respectively. These offsets are added to the current coordinates of CanDoMan. **QueryInteraction** is called to get a HANDLER for the destination square, and the HANDLER is invoked.

Kill_Player

This Routine operates just like the Bot **Kill** Routines, with the additional check to see if the player has run out of lives, and if so, a sound is played and the **EndGame** Routine is called to stop game play.

Erase_Player

This is essentially the same Routine as used by the Bots.

Show_Player	See above.
KeyFlag_Player	See above.

And finally, there are routines just to handle the flow of the game:

InitAll	This Routine loads all the player and bot images, sets up the first level, sets the score to 0, and initializes CanDoMan's lives.
EndGame	Shuts down all Bots, turns GAMEINPROGREESS variable to NO, and deactivates the Timer Routine.
MagicCount	Updates the color of the magic wand based on the MAGICCOUNT variable.
NewGame	Ends the current game, sets up the sounds to play and redisplays level 1, then starts up the bots and places the player on the display.
NextLevel	Moves game to the next level and changes the speed of the timer events.
NormalTick	The Routine which is called on each Timer event when the game is in play. It increments the COUNTER and when it reaches MAXCOUNT it moves the Bots.
NothingTick	The Routine which is called on each Timer event when the game is not in play.
PauseGame	Switches between Paused and UnPaused without changing the score and players status.
ResetLevel	Sets up variables to indicate the first Level of play.
ResetLives	Gives the user a full LIVES count, and redisplays this count.
ShowKey	Place the image of the Key on the display.
ShowLevel	Displays which level you are currently on.

ShowLevelImage	If we have switched levels, display the new level imagery.
ShowLives	Update Lives count if it is changed, and redisplay that count on the screen.
ShowScore	Update the score count and then redisplay it on the screen.

"CodeBuster"

How To Play CodeBuster

CodeBuster is a game of logic. The computer has a code of four colors. Your mission is to determine what that code is.

There are six colors available. The computer selected four times from among these six colors. You can enter a guess as to what four colors the computer has by clicking on a color, then clicking on one or more of the four balls in the recessed area near the top of the screen. This will put the color you have chosen on each ball you select. When you have set any or all of the colors, select the Compare button. The computer will then tell you how many colors are in the right position (the number will be displayed in white) and how many colors that are correct, but NOT in the right position (displayed in black).

For example:
The computer has chosen red, green, blue, red.

Your choices:	Computer responds:
red, blue, green, yellow	1 2
blue, green, yellow, orange	1 1
green, red, red, blue	0 4
red, green, blue, red	4 (Got it!)

You can click on almost anything on the screen to select a color, except for the four balls where you are supposed to be entering colors (the balls in the recessed area). This means that you can select colors from the balls that scroll below the top row of balls. The mouse pointer will change to the color you select. Try it; you'll see what we mean. If you don't enter a new color in a position, when you select Compare the previous color will be used for your guess. Initially, these colors are red, blue, green, and yellow, respectively, and are displayed as an initial guess for your convenience.

If you want to see the answer, press the Right-Amiga-S combination. For a new game, press Right-Amiga-N. To turn the color-change feature on/off, press Right-Amiga-C. A brief help screen will be displayed if you press Right-Amiga-H. To quit, press Right-Amiga-Q. These options can also be selected from a menu.

How CodeBuster Works

CodeBuster only has one Card. The background color is filled into the Window in the AfterStartup Script by extensive color manipulation. This is also where the colors for the four buttons is decided, and they are stored in the variables named COLOR1 through COLOR4. Only color numbers 3 through 8 are used for the 6 colors from which the computer picks its secret code. There are a number of Buttons on the Card. They are:

ColorButton This is a big Button which covers the entire window. Anytime the user clicks anywhere in the window, the

OnClick script of this Button will sample the color of the window underneath the mouse pointer at that time. If the color is less than MINCOLOR (3) or greater than MAXCOLOR (8), then it is not a valid color for one of the balls, so the script does nothing in those cases. If the color is within range, the color is saved in the variable THISCOLOR, and the pointer colors, which are always colors 17 through 19, are set to that color so that the pointer is solidly that color.

BoxButton — In order to prevent the user from being able to choose colors from the recessed area, this Button is set on top of the "ColorButton" and does nothing when hit.

Button1 - Button4 — For each of the four balls in the recessed area, when they are picked, if there has been a color selection so far, the ball is redrawn in the new color, and the value of the THISCOLOR variable are stored in the appropriate MYCOLOR variable. The first ball's color is stored in the variable MYCOLOR1, the second ball's color in MYCOLOR2, and so on.

Compare — When the user is ready to compare his or her guess to the actual secret colors of the balls, this Button's **OnRelease** script gets activated. The script follows these steps:

1) The four secret colors are typed into a Document called "CColors", one per line. Those colors are stored in the variables called COLOR1 through COLOR4.

2) The colors from the four balls in the recessed area are typed into a Document called "MyColors", one per line. Those colors were stored in the MYCOLOR set of variables. These are the user's guessed colors.

3) The colors in each Document are compared, one at a time. When two colors match, that means that the player has correctly guessed the color of one of the balls. The variable CORRECTCP is increased by one, and the matching lines are replaced in both Documents with a value of 0, so that when a check is made for the right color in the wrong slot later on, these colors will not be considered again. Remember, only colors 3 through 8 are valid colors for the balls.

4) If all four of the user's guesses were correct (CORRECTCP=4), then the Routine **GotItRight!** is called and does not return.

5) The Document containing the user's guessed colors is searched, line by line, for each of the remaining true colors. If they are found, that means that the user has the correct color for a ball, but in the wrong position. The variable CORRECTCOLOR is used to count the number of times this condition is discovered.

6) The inside of the recessed area is clipped into a Brush Buffer called "ClippedBrush". Next, a rectangular area immediately below the recessed area is

scrolled down, 2 pixels at a time, for a total of 12 pixels. In this way previous guesses are moved down the screen. Then the most recent guess inside "ClippedBrush" is stamped onto the display with a **ShowBrush** command.

7) Finally, the number of correct colors/positions, and correct colors/incorrect positions, are printed into the window next to the most recent guess.

NewGame This Button is simplicity itself. It simply performs a **FirstCard** command, effectively restarting the Deck.

The CodeBuster Deck also has a few Routines:

Ellipse Each time a ball is drawn anywhere in the window, this Routine draws the ball, puts the little white area on it to make it look shiny, and draws the shadow area underneath it to complete the illusion of a light source shining down on the ball from the upper left.

GotItRight! When the user finally wins, the "Compare" Button gets disabled, and the various text messages are printed to the screen indicating the level of achievement of the user. The variable TRIES is used to keep track of the total number of guesses that the user has made. If TRIES is less than 10, the colors are also cycled briefly for a little additional flair.

Print This little Routine is called by the **ShowInfo** Routine when it is explaining the rules. The text to print is printed and a variable is used to keep track of the position at which to print next.

ShowCurrent This Routine draws the four balls in the recessed area. It is called from the AfterStartup script of the Card in order to show the starting colors, and is called whenever the user gives up via the "Show Answer" TextMenu item. In that case, the MYCOLOR set of variables is modified to be equal to the COLOR set of variables so that the colors shown are the correct ones.

ShowInfo The instructions for CodeBuster are printed via this Routine.

"The PaintBox"

How To Use The Paintbox

The PaintBox Deck is a small paint program made using CanDo. A paint program is one which anybody can use to draw pictures, or touch up pictures that others have drawn. On the Amiga, paint programs are very popular because of the vast number of colors and flexible resolution it offers, and because of the simplicity of drawing with a mouse.

You can run PaintBox from the Workbench or the CLI, however if you run it from the CLI you must be logged into the same directory as the PaintBox Deck. After PaintBox starts, you will have a blank screen onto which you can paint. Every option of the PaintBox is controlled through pull-down menus, the mouse, and the keyboard.

Try clicking the left mouse button on the PaintBox screen, and drag the mouse and then release the left button. You have just drawn a dotted line! You can change the type of thing you are drawing through the "Tool" menu. If you select this menu, you will see the following choices:

Dotty
Doodle
Line
Flood Fill
Open Rectangle
Solid Rectangle
Open Circle
Solid Circle
Open Ellipse
Solid Ellipse

By choosing one of these menuitems, you will immediately change the type of drawing that you are doing.

To change drawing color, press the "P" key. This will cause a palette of colors to open at the top of the screen. You can select one of these colors by simply clicking on it. To do so will immediately change which color you are drawing with. Don't worry about the fact that the palette is on the screen, in your drawing area: anytime you start to draw it will disappear and will reappear whenever you release the left mouse button. To make the palette go away, press the "P" key again.

Under the "Clip" menu, you will find commands that allow you to load and save brushes, and to clip brushes from the display. A brush in PaintBox is a part of any image with which you can draw onto the screen instead of using the thin line with which you normally draw. PaintBox will be able to use any brushes made with other Amiga paint programs, such as DPaint III. The "Clip" menu offers the following commands:

(cont. next page)

Open...
Save
Save as...
Grab A Clip
Use Clip

Open... will bring up the built-in CanDo file requester, allowing you to select a brush with which to paint. After selecting a brush, the **Use Clip** command will allow you to draw with it.

Save will save the current clip brush using the same name as the most recent **Save** or **Open...** command used. If no clip filename has yet been used, the CanDo file requester will appear. The brush will be saved as a standard, DPaint compatible brush file.

Save as... lets you specify a new name for the clip via the file requester, and will then save the clip brush under that name.

Grab A Clip allows you to specify a section of the screen to use as a clip brush for drawing. After selecting this option, click and drag in the screen with the mouse. This action will select the clip area. When you release the mouse button, the clip will be stored internally.

Use Clip will let you draw with the currently loaded, or last clipped, clip brush. This type of drawing is very similar to that used by other Amiga paint programs. As you click and drag the mouse, the clip brush will be drawn onto the screen.

Finally, under the "Project" menu, you will find the following options:

Refresh Screen
Clear Screen
Open...
Save
Save as...
Quit

Refresh Screen will redraw the original picture that was loaded from disk. In the event that the picture is no longer in memory, PaintBox will try to re-load it. If it cannot be re-loaded, PaintBox will give an error message. This is not an undo feature; it simply redraws the original screen.

Clear Screen will erase everything you have drawn into the display area. Be careful, there is no undo for this command.

Open... allows you to load in a picture file of the type that DPaint or another Amiga paint package might create. This picture will then be displayed in the drawing area. You can load pictures and touch them up using PaintBox in this way.

Save will save the current display as a standard Amiga picture file (ILBM in IFF terms) to the filename used last by the **Open...** or the **Save**, or **Save as...** commands. If

a name has not yet been specified, the file requester will open, allowing you to pick a filename.

Save as... will open the CanDo file requester to let you pick a name to which to save the current display, and will then perform the save function after you have picked a filename.

That's it; a simple paint program, created with CanDo. Let's see how it does what it does.

How PaintBox Works

Basically, the PaintBox Deck uses a Picture Window, completely covered by a big Area Button. The **OnClick**, **OnDrag**, and **OnRelease** scripts for the Area Button accomplish all of the drawing functions through the use of *indirection*. Indirection is a method of programming whereby the contents of a variable completely or partially contains the name of some other part of the program. For example, suppose you want to **Do** one of two possible Routines, based on a variable **X**, which can be either 1 or 2. In CanDo, you might do it like this:

```
If X=1
    Do "X1"
Else
    Do "X2"
EndIf
```

which would work fine. However, a shorter and simpler way of accomplishing this same goal is shown in the following example.

```
Do "X" || X
```

In this case, if **X** equals 1, the Routine name to **Do** will be evaluated to be "X1", and if **X** equals 2, the name will be evaluated to be "X2". This is indirection at work, and it is used extensively in the Decks on the ProPack 1 diskette. In the case of the PaintBox, indirection is used to allow the scripts for the one Area Button to perform several drawing functions. Each time a new drawing tool is selected from the "Tools" menu, several variables are set to values associated with that tool. When the Area Button is accessed by the user, these variables are used to figure out which Routines to call to perform the functions.

The PaintBox Deck consists of two Cards:

BlankImage	This first Card opens an extra-halfbrite screen, sets up its palette, clips the screen into a Picture Buffer using the **ClipPicture** command, and then goes to the next Card. By using this method, the "RealPainter" Card can be set up as a Picture Window without having to have a real picture on the disk.
RealPainter	All of the interactive parts of the program happen on this Card. Basically, there is one, very large Area Button (640 by 400)

which covers the screen. Anytime the user clicks on this Button, the appropriate Routines are called to perform the type of drawing required by the current drawing tool.

The "RealPainter" Card contains the main Area Button for the drawing area, the Area Button on which the palette selector resides, as well as all of the TextMenu Objects which are used to control the program. A single **AKey** Object is also defined on this Card to allow the user to "open" and "close" the palette selector. (All that really happens is that the palette Area Button is moved so that it is in or out of the visible area.)
The only Object here of any real complication is the Area Button which encompasses the drawing area. Here is the complete definition of this Object:

```
AreaButton "DrawArea"
    Definition
        Origin 0,0
        Size 640,400
        Border NONE
        Highlight NONE
    EndDefinition
    OnClick
        Do "HideColorBox"
        Do PLACEMARK
        IfError
        EndIf
    EndScript
    OnRelease
        Do LIFTMARK
        IfError
        EndIf
        Do "RefreshColorBox"
    EndScript
    OnDrag
        Do USEMARK
        IfError
        EndIf
    EndScript
EndObject
```

In each script of the Object, indirection is used to activate the appropriate Routine for the currently selected tool. When the mouse is clicked on the Area Button, the **PLACEMARK** script is called to start drawing (think of this as setting a pen down on paper.) When the mouse is dragged, the **USEMARK** script is called to actually draw on the screen and keep up with the mouse (moving the pen.) Finally, when the mouse button is released, the **LIFTMARK** script is invoked to complete the drawing activity (lift the pen off the paper.)

Each TextMenu Object in the "Tools" menu provides the Routines to accomplish one drawing function, and sets the **PLACEMARK**, **USEMARK** and **LIFTMARK** variables to values appropriate for that function. They do this by calling either the "UsePen" or the "UseTool" Routine, passing to it the name of the drawing function as

ARG1. "UsePen" is called for drawing functions which behave like real pens, i.e., you put them on paper, drag them around, and wherever the pen touches the paper, something is permanently drawn. The "UseTool" routine is called for each drawing functions that must "rubber-band" around the display. For example, the rectangle tools will follow the mouse around until the user lets up on the mouse button. Only then will the display be permanently altered. "UseTool" uses a set of three Routines, "PlaceTool", "DragTool" and "LiftTool" to accomplish the rubber-band effect.

Not all tools require Routines for all three functions. That is why the **IfError...EndIf** commands are used after the calls to the **PLACEMARK**, **USEMARK** and **LIFTMARK** Routines. If the particular Routine does not exist, the **Do** command will generate an error. By using **IfError** these errors will be ignored when the occur.

Notice that in the **OnClick** script the "HideColorBox" Routine is called, and that in the **OnRelease** script "RefreshColorBox" is called. If the palette requester is in the visible display area, these two calls will move it out of the way while the user is drawing, and will move it back when the user stops drawing.

These are the Routines used in the PaintBox Deck:

Doodle The PLACEMARK Routine for the Doodle tool.

Doodle.Drag The USEMARK Routine for the Doodle tool.

Dotty The PLACEMARK Routine for the Dotty tool.

Dotty.Drag The USEMARK Routine for the Dotty tool.

DragTool When a rubber-banding tool is in effect and the mouse is dragged, this Routine is called. It draws the image in COMPLEMENT mode, effectively erasing the old image, rescales it according to the new mouse position, and then draws it again.

FloodFill The only Routine required by the FloodFill tool, this is its PLACEMARK Routine.

GrabClip The GrabClip function uses the OpenRectangle tool's Routines to show the sizing of the clip area.

GrabClip.Release When the user lets up on the mouse button after sizing the clip area, this Routine figures out how big the clip is and uses the **ClipBrush** command to save it in a Brush Buffer named "TheClip."

HideColorBox The palette selector, or color box, is moved out of the visible display area with the **MoveObject** command, and what was previously in its place is restored. This happens only if the color box is currently being shown, and the user has started to draw on the screen. When the user lets up on the mouse button, the "RefreshColorBox" Routine is used

to restore the palette selector on the display.

InitAllResources The palette selector is really a brush that is loaded off disk. A different brush is required for every possible number of colors that might occur on a screen, i.e., 2, 4, 8, 16, 32, 64, or 4096 colors. Each time a new picture is selected, or the "Refresh Screen" or "Clear Screen" commands are used, this Routine is used to make sure that the proper color box brush is loaded into memory.

LiftTool When a rubber-banding tool is in effect and the mouse is released, this Routine is called. It draws the image in NORMAL mode, making it a permanent part of the display.

Line This is the **TOOL** Routine for drawing lines. This tool uses rubber-banding.

OpenCircle This is the **TOOL** Routine for drawing open circles. This tool uses rubber-banding.

OpenEllipse This is the **TOOL** Routine for drawing open ellipses. This tool uses rubber-banding.

OpenRectangle This is the **TOOL** Routine for drawing open rectangles. This tool uses rubber-banding.

PlaceTool When a rubber-banding tool is in effect and the mouse is released, this Routine is called. It calculates the current mouse position and the initial size of the image to draw, and draws the image once in COMPLEMENT mode. This sets up the necessary environment for the "DragTool" Routine to function.

RefreshColorBox Whenever the color selector is moved into the visible display, the area which will be underneath the color box is clipped into a Brush Buffer called "BoxClip", so that it can be restored whenever the color box is removed.

RemoveColorBox This Routine goes through the same steps as the "HideColorBox" Routine, with the additional step of setting the **COLORBOXSHOWN** variable to false. This prevents the palette selector from being shown in the future, until the user deliberately opens it again. This is the Routine that is used to shut down the color box when it is already open and the user presses the "P" key.

ReportError If errors are encountered while performing various functions, PaintBox will call this Routine, giving it a string to echo out to the CLI. If PaintBox was not run from a CLI, this string will not be seen.

This brings up the file requester and gets a name under which to save the current clip. If it gets a valid filename, it will save the clip and store the filename in the variable **CLIPNAME**. This checks first to make sure that a clip exists, and will do nothing if no clip has been created or loaded.

SaveAsPicture This brings up the file requester and gets a name under which to save the current display. If it gets a valid filename, it then saves the picture and stores the filename in the variable **PICNAME**.

SaveClip This Routine saves the current clip, if there is one, using the filename stored in the **CLIPNAME** variable.

SavePicture This Routine saves the current display using the filename stored in the **PICNAME** variable.

SelectColor Whenever the user clicks on the palette selector, this Routine checks to see if a new color has been chosen, using the **ColorOfPixel** function. If a new color has been selected, it changes PenA to be that color, and then calls "UpdateColorBoxColor" to adjust the currently selected color display at the far left side of the color box.

ShowColorBox When the user presses the "P" key, and the color box is not currently being shown, this Routine is called. It sets the **COLORBOXSHOWN** variable to true and then calls the "RefreshColorBox" Routine to force the palette selector into the visible area.

SolidCircle This is the **TOOL** Routine for drawing solid circles. This tool uses rubber-banding.

SolidEllipse This is the **TOOL** Routine for drawing solid ellipses. This tool uses rubber-banding.

SolidRectangle This is the **TOOL** Routine for drawing solid rectangles. This tool uses rubber-banding.

UpdateColorBoxColor Each time the palette selector is placed on the display, or a new color is chosen by the user, this Routine draws a rectangle into the color box at the far left side, showing which color is the current drawing color.

UpdateMove This updates the mouse position variables, and rescales the size of the rubber-band area. In this way, the tools requiring a rubber-band effect, can keep up with the mouse as it is dragged.

UseClip	The **PLACEMARK** Routine for the Clip tool.
UseClip.Drag	The **USEMARK** Routine for the Clip tool.
UseClipPen	When "Use Clip" is chosen from the "Clip" menu, this Routine checks to see if there is a clip, and if there is not, it sets up the use of the Dotty tool instead.
UsePen	This is the setup Routine for all tools which do not require a rubber-band effect.
UsePointer	Each tool has a unique pointer to remind the user which paint function is being used. This Routine sets the pointer to the required image. If the pointer brush cannot be found, the error will be ignored.
UseTool	This is the setup Routine for all tools which do require a rubber-band effect.

How To Add New Drawing Functions

Suppose that we wanted to add a new drawing tool to PaintBox, e.g., an Open Square tool. We would proceed like this:

1. Add a menuitem for "Open Square" to the "Tools" menu by duplicating the "Open Rectangle" Menu Object. Change its **Occurred** script to read:

   Do "UseTool","OpenSquare"

2. Duplicate the "OpenRectangle" Routine, and change it to read:

   DrawRectangle STARTX,STARTY,WIDTH,WIDTH

   so that the height of the rectangle that is drawn is the same as the width.

3. Finally, create a new brush for the Open Square tool, or just copy the Open Rectangle pointer file. Remember, you can use the PaintBox itself to create the file for this new pointer image!

Layout EditorTool

Introduction

With the Layout EditorTool, you can integrate text into your CanDo displays more easily than ever before. This tool allows you to flow text, which you can load from a file or type in yourself, around imagery or other display elements. You simply use the mouse to guide the boundaries of the text layout area, and the script to exactly reproduce your layout will be written for you. You may also specify the font and style in which you want the text rendered.

Installation

To install the Layout EditorTool, just move the entire directory named Layout into your EditorTools directory used by CanDo. If you are working with the original CanDo distribution floppies, there may not be enough room for the directory on the CanDoExtras disk. One solution might be to move the entire EditorTools directory to RAM, but wherever you put it, be sure to update the EditorTools ToolType in the CanDo icon or the cando.defaults file in the "s" directory on the CanDo disk.

To move the entire Layout directory using Workbench, drag the Layout icon onto the EditorTools icon. The copy will be performed and will take only a few moments. From CLI just type

"copy Layout [wherever]:EditorTools all"

and the copy will be performed.

Using The Layout EditorTool

Layout is an EditorTool; you use it while in CanDo's script editor. After successful installation, it will appear as an icon in the scrolling list of EditorTools at the far right side of the script editor. You may have to scroll through the list using the slide bar to find the Layout icon. It looks like this: 

Click once on the Layout icon to run it. After a few seconds, it will open a window on CanDo's screen called "Layout Editor Tool", and will also draw a default *shape* on your window. The shape is the text layout area. Whatever text you layout will be flowed into this area.

The Layout EditorTool window has a text editor into which you can type text, or load a text file. This tool has one menu called "Text" which has three menuitem choices:

Load...
Save...
Insert...

Load... will open the CanDo file requester and allow you to specify a text file to load into the Layout editor. After loading the text, you may type changes into it. Layout

uses the text that is in the editor when it flows the text onto your display. It does not read the text from the file ever again.

Save... lets you save whatever text is in the Layout editor to a text file of your choosing.

Insert... will add the contents of whatever text file you specify to the text that is currently in the Layout editor.

Type some text into the Layout editor, or load a small text file. At any time, you can see what the text will look like once it has been flowed by pressing the button labeled "Preview the text" at the right side of the display. When you press this button, a requester will open which tells you how to stop previewing the text and return to the Layout EditorTool control window. Pay careful attention to this information. To return to the Layout control window you will press the Escape key once. After you press the OK button on this requester, Layout will start to calculate the best way to flow your text. It will use an hour-glass pointer to show you how it is progressing. After a few seconds, your text will appear in the flow area on your window.

To adjust the flow area, choose one of the three buttons at the top of the Layout window. These buttons are labeled:

Left
Top
Right

By using these three options, you can modify the boundaries of the flow area, directly on your display. After choosing one of the buttons, the Layout control window will move out of your way and the pointer will turn into an arrow showing you which side of the flow boundaries that you are editing. You can switch between the three directions by using the arrow keys on your keyboard. By pressing the up arrow, for example, will change your pointer to an up arrow and you will then be in the proper mode to modify the top boundary of the text flow area. To return to the Layout control window you will press the Escape key once.

Left allows you to change the left-hand boundary of the text flow area. Notice that the left and right boundaries are made up of vertical slashes. By clicking on the display and dragging your mouse while in left arrow mode, you will move the individual slashes on the left-hand side of the flow area. You may have to drag the mouse slowly to move each and every slash to the correct position.

Top allows you to set the top boundary for the text flow area. Just click in your window and drag the mouse. A horizontal line will appear that represents the top boundary. Release the mouse button when you have the boundary at the proper spot.

Right lets you set the right-hand boundary of the text flow area in the same manner as **Left**.

Remember to press the Escape key to return to the Layout control window.

You can also control the font and text style of the flowed text by pressing the button

labeled "Set Font & Style". This will open the CanDo Font/Text Requester from which you can specify the font and point size to use, as well as the style, including CanDo's special extended text styles.

After setting the font and style, Layout will recalculate the size of the flow area. This will take only a few seconds.

When you have setup your flow area, and the text to go in it, just the way you want it to be, press the "OK" button, and Layout will type all of the CanDo scripting commands necessary to display your text within the flow area. If you have a large amount of text, this can produce a very large script.

It is a good idea to save your text before pressing OK, if the contents of the Layout editor are not already saved. The Layout EditorTool cannot work backwards from a script to calculate what your text is or what the shape of the flow area is.

KeyInput Object XtraTool

Introduction

The KeyInput Object is an Object which you may use in your CanDo applications for the purpose of responding to keyboard activity.

Three types of events are associated with KeyInput Objects: Pressed, Repeated and Released. A script written for the Pressed event will execute when the designated key is pressed down. A script written for the Repeated event will execute regularly as the designated key is held down. And a script written for the Released event will execute when the designated key is finally released.

NOTE: The designated key may also involve command keys, such as SHIFT and CONTROL, to be held down when the key is pressed, as will be explained below.

Installation

To install the KeyInput Object, just move the entire directory named KeyInput into your XtraTools directory used by CanDo. If you are working with the original CanDo distribution floppies, there may not be enough room for the directory on the CanDoExtras disk. One solution might be to move the entire XtraTools directory to RAM, or to the CanDo disk (you might have to delete some other files from the CanDo disk to do so), but wherever you put it, be sure to update the XtraTools ToolType in the CanDo icon or the cando.defaults file in the "s" directory on the CanDo disk.

To move the entire KeyInput directory using Workbench, drag the KeyInput icon onto the XtraTools icon. The copy will be performed and will take only a few moments. From CLI just type

"copy KeyInput [wherever]:XtraTools all"

and the copy will be performed.

Using The KeyInput Object

From the main panel, press on the button marked "Xtra." A requester titled, "Extra Objects/Tools System," will appear. Position on the KeyInput line and select the "Perform" button. The "KeyInput Object System" requester will appear. Select the "Add" button to create a KeyInput Object. If the KeyInput Object System was properly installed, you will now see the "KeyInput Object Editor." If not, repeat the installation procedures above and try to use the KeyInput Object again. If problems persist, call the INOVAtronics CanDo Help Line at (214) 340-4992.

Three Fields in the "KeyInput Object Editor" must be filled out for each KeyInput Object you create. The first Field, "Name," is the name of the KeyInput Object you are creating and can be anything you wish.

The "Key Code" Field contains the code for the actual key which will trigger an event. The following Key Codes are valid for this Field:

0	P	CAPSLOCK	NUM7
1	Q	CLOSEBRACK	NUM8
2	R	COMMA	NUM9
3	S	COMMODORE	NUMASTERISK
4	T	CONTROL	NUMCLOSEPAR
5	U	DELETE	NUMHYPHEN
6	V	DOWN	NUMOPENPAR
7	W	ENTER	NUMPERIOD
8	X	EQUAL	NUMPLUS
9	Y	ESCAPE	NUMSLASH
A	Z	HELP	OPENBRACK
B	F	HYPHEN	PERIOD
C	F2	INTER1	RETURN
D	F3	INTER2	RIGHT
E	F4	LEFT	RIGHTALT
F	F5	LEFTALT	RIGHTAMIGA
G	F6	LEFTAMIGA	RIGHTSHIFT
H	F7	LEFTSHIFT	SEMICOLON
I	F8	NUM0	SINGLEQUOTE
J	F9	NUM1	SLASH
K	F10	NUM2	SPACE
L	ACCENT	NUM3	TAB
M	AMIGA	NUM4	UP
N	BACKSLASH	NUM5	
O	BACKSPACE	NUM6	

The "Command Keys" Field contains the set of keys (up to three) which must also be held down when the key specified in the "Key Code" Field is pressed. If more than one Command Key is desired, each must be separated by a space. If you do not require any Command Keys, then enter NONE into the "Command Keys" Field.

These five command keys can be used in any sequence:

CAPSLOCK	LEFTMOUSE	RIGHTMOUSE
CONTROL	MIDDLEMOUSE	

along with one command key from each of the four groups below:

SHIFT KEYS	ALT KEYS	LEFT AMIGA	RIGHT AMIGA
BOTHSHIFTS	ALT	COMMODORE	AMIGA
LEFTSHIFT	BOTHALTS	LEFTAMIGA	RIGHTAMIGA
RIGHTSHIFT	LEFTALT		
SHIFT	RIGHTALT		

Once you have selected the Key Code and the Command Keys for the KeyInput Object, you can create the scripts for the events you wish to monitor. When finished, select the "OK" button and then the "Exit" button to return to the Main Panel. To test your newly added KeyInput Object, select "Browse." You can now use the KeyInput Object you just created.

One circumstance which will prevent any KeyInput Objects from functioning has to do with Field Objects. If a Field Object is currently activated, i.e., has the cursor in it, then KeyInput Objects will not be processed by CanDo, because the Field is receiving all of the input from the keyboard. Once the Field is no longer activated, normal KeyInput Object handling will resume.

The KeyInput Object XtraTool is only required for creating and editing KeyInput Objects. It is not required for running the created applications. In order to edit the keys used in the CanDoMan game, you will need to have the KeyInput Object XtraTool installed.

CrossRef

Introduction

The CrossRef program is a utility which can analyze a given Deck and tell you the following kinds of information:

The names of all Cards

The names and types of all Objects

The names of all Routines

Usage counts of all variables

All instances of references to particular variables, functions, strings or expressions

Every usage of particular commands

All instances of the use of indirection or other unresolved references

Much of the analysis can be limited to individual Cards rather than all Cards in the Deck.

The information you gather with the CrossRef program can be used to find and fix programming errors in your Decks, facilitate making global changes to Buffer or Object names, and assist in providing an overall feel for what the Deck is doing. We regard it as an invaluable complement to the functions of ThePrinter utility.

Results from cross-reference analysis can be sent to a disk file or to a printer.

Installation

The CrossRef utility can be installed to anywhere in your working environment. It makes sense to locate it in your CanDo work directory if you have a hard drive. If you are using floppies, you can just leave it where it is, on a backup copy of the ProPack1 diskette.

Another program must be present in the C: directory before CrossRef will work, however. The file DeckCrossRef can be found in the same directory with CrossRef, and since it has no icon, you must copy it using the Copy command from a CLI or Shell. Open a CLI or Shell and type the following command:

Copy ProPack1:TheCrossRef/DeckCrossRef C:

This operation will take a few seconds to complete. If your disk becomes full during the copy, see what files you may be able to eliminate from the C: directory. It is required that DeckCrossRef be installed in the C: directory before the CrossRef utility will work.

It is also required that cando.library be in the LIBS: directory of your system disk, or that it already be running and installed in memory. Running CrossRef without cando.library being available will cause a requester to open, letting you know that the library is required.

After installing the DeckCrossRef and the CrossRef programs, you may run the CrossRef program from Workbench by double-clicking its icon, or you can run it from a CLI or Shell by typing its full path and filename.

Using The CrossRef Utility

Once you have the CrossRef utility running, the next step to take is to load a Deck to examine. Click on the words "Deck Name", at the far left of the CrossRef window, and a file requester will appear to help you select a Deck to load. This is the same file requester as the one used in CanDo, and its operation is identical. Use it to choose a Deck to examine, and press its OK button. The selected filename will appear in the field immediately to the right of the words "Deck Name".

When the first operation is performed that will require analysis of the Deck you have named, the Deck will be loaded. If it is not a valid CanDo Deck, or cannot be found, an error message will be displayed in the message area at the top left of the CrossRef window.

The commands that you can choose from for getting information on a Deck are listed in the requester area at the top right of the CrossRef window. This is a scrolling list of commands, and to invoke any command you must double-click on the command name in this list.

Some commands require that the field immediately below this list be filled in with additional qualifiers. These are the commands which have the symbol, "{?}", somewhere in the body of the command name in the list. The field to be filled out is marked with the symbol, "{?}", immediately to its left. An example of a command of this type is **UsageOf Command {?}**. In this case, CrossRef will locate every instance of the use of a particular command, as specified by the contents of the "{?}" field.

For example, to search for each usage of the CanDo scripting command **GotoCard**, just type "GotoCard" into the "{?}" field and double-click on the **UsageOf Command {?}** command from the list. After a few seconds, information will appear in the scrolling list which occupies the bottom of the CrossRef window. This is the Results Area, and this list is where the results from all of your various inquiries about your Deck are stored. This list can be printed to a printer or to a disk file. To print the list, just type the name of the file or printer to which to print in the field next to the "Print" button. (This field is immediately below the "Deck Name" field.) Use "PRT:" in this field if you want to send the output to a printer, otherwise type in a valid filename. Each time you press the "Print" button, the contents of the Results Area will be saved to the print file, or sent to the printer.

Note that the Results Area is also a Memo; you can type anywhere into the Results Area in order to make notes to yourself about the details disclosed by your inquiries.

To get a feel for the type of information that the CrossRef utility can yield, choose the

List Objects command from the command list. Each Object in your Deck will be listed in the order in which it is found. In this case, each Object's detail will contain the following information:

The name of the Card on which the Object exists
The Object type
The name of the Object

At any time, you can limit the scope of your inquiries to individual Cards. You do this by typing the name of the Card that you want to search into the field called, "Limit References To". You can also double-click on any word in the Results Area, and it will be placed into the "Limit References To" field automatically. When searches are limited to a given Card, the Card name will be omitted from the results that are placed into the Results Area. To cancel any limitations on searches, simply clear the "Limit References To" field.

Instructions for each CrossRef command follows.

List Cards will place the names of all Cards in the Deck into the Results Area, in the order in which the Cards exist in the Deck.

List Objects lists every Object within the scope of the search (either every Card, or just the Card named in the "Limit References To" field. The Objects will be listed in the order in which they exist on their respective Cards.

List Routines will name every Routine defined in the Deck, in the order in which they are defined. Information on each Routine will consist of:

The name of the Routine

List Variables (usage) creates a list of all variables used in your Deck, and orders them by their frequency of use. Below is an example variable listing from the CodeBuster Deck on the ProPack1 disk.

```
*********
* List Of Variables in all cards. (usage count)
*********
* Usage Name
*    1 ARG1
*    1 COLOROFPIXEL
*    1 MOUSEX
*    1 MOUSEY
*    2 BC
*    2 GC
*    2 RC
*    3 COLOR1
*    3 COLOR2
*    3 COLOR3
*    3 COLOR4
*    3 REVEALED
*    4 B
```

```
*    4 CORRECTCOLOR
*    4 G
*    4 R
*    4 TEMPCOLOR
*    4 THELINE
*    5 MYCOLOR1
*    5 MYCOLOR2
*    5 MYCOLOR3
*    5 MYCOLOR4
*    5 X
*    6 ARG2
*    6 ARG3
*    6 CORRECTCP
*    8 B1
*    8 G1
*    8 J
*    8 R1
*    9 TRIES
*   10 RANDOM
*   11 MAXCOLOR
*   11 MINCOLOR
*   14 THISCOLOR
*   20 I
*********
```

The number preceding each variable name is the number of times that the variable is used in the Deck. This command sorts the list of variables so that the most frequently used variables appear at the bottom of the list, and the least used variables appear at the top. The list of variables includes usage of the variables that you have created for use inside your Deck as well as CanDo system variables (e.g., MOUSEX and MOUSEY) and CanDo functions (COLOROFPIXEL and RANDOM).

List Variables (alpha) creates a list of the same variables as the previous command, however this list is sorted alphabetically by variable name.

UsageOf Command {?} requires that the "{?}" field be filled out with the name of a valid CanDo command, such as ***Do*** or ***GotoCard***. Given that a CanDo command name is entered into the field, this command will create a list of all usages of the command within the scope of the search as defined above. What follows is an example search for the command **Do** in the CodeBuster Deck from ProPack1.

```
*********
* Usage of Command "Do" in whole deck.
*********
* Routine "ShowCurrent"
* Line Number: 1
* -> Do "Ellipse",MyColor1,65,35
*
* Routine "ShowCurrent"
* Line Number: 2
* -> Do "Ellipse",MyColor2,86,35
```

```
*
* Routine "ShowCurrent"
* Line Number: 3
* -> Do "Ellipse",MyColor3,107,35
*
* Routine "ShowCurrent"
* Line Number: 4
* -> Do "Ellipse",MyColor4,128,35
*
* Card "CodeBustCard"
* Object Type: Card
* Object Name: "CodeBustCard"
* Script Name: AfterStartup
* Line Number: 43
* -> Do "Ellipse",3,43,5
*
* Card "CodeBustCard"
* Object Type: Card
* Object Name: "CodeBustCard"
* Script Name: AfterStartup
* Line Number: 44
* -> Do "Ellipse",4,64,5
*
* Card "CodeBustCard"
* Object Type: Card
* Object Name: "CodeBustCard"
* Script Name: AfterStartup
* Line Number: 45
* -> Do "Ellipse",5,85,5
*
* Card "CodeBustCard"
* Object Type: Card
* Object Name: "CodeBustCard"
* Script Name: AfterStartup
* Line Number: 46* -> Do "Ellipse",6,106,5
*
* Card "CodeBustCard"
* Object Type: Card
* Object Name: "CodeBustCard"
* Script Name: AfterStartup
* Line Number: 47
* -> Do "Ellipse",7,127,5
*
* Card "CodeBustCard"
* Object Type: Card
* Object Name: "CodeBustCard"
* Script Name: AfterStartup
* Line Number: 48
* -> Do "Ellipse",8,148,5
*
* Card "CodeBustCard"
```

```
* Object Type: Card
* Object Name: "CodeBustCard"
* Script Name: AfterStartup
* Line Number: 58
* -> Do "ShowCurrent"
*
* Card "CodeBustCard"
* Object Type: AreaButton
* Object Name: "Button1"
* Script Name: OnRelease
* Line Number: 5
* -> Do "Ellipse",ThisColor,65,35
*
* Card "CodeBustCard"
* Object Type: AreaButton
* Object Name: "Button2"
* Script Name: OnRelease
* Script Name: AfterStartup
* Line Number: 44
* -> Do "Ellipse",4,64,5
*
* Card "CodeBustCard"
* Object Type: Card
* Object Name: "CodeBustCard"
* Script Name: AfterStartup
* Line Number: 45
* -> Do "Ellipse",5,85,5
*
* Card "CodeBustCard"
* Object Type: Card
* Object Name: "CodeBustCard"
* Script Name: AfterStartup
* Line Number: 46* -> Do "Ellipse",6,106,5
*
* Card "CodeBustCard"
* Object Type: Card
* Object Name: "CodeBustCard"
* Script Name: AfterStartup
* Line Number: 47
* -> Do "Ellipse",7,127,5
*
* Card "CodeBustCard"
* Object Type: Card
* Object Name: "CodeBustCard"* Script Name: AfterStartup
* Line Number: 48
* -> Do "Ellipse",8,148,5
*
* Card "CodeBustCard"
* Object Type: Card
* Object Name: "CodeBustCard"
* Script Name: AfterStartup
```

```
* Line Number: 58
* -> Do "ShowCurrent"
*
* Card "CodeBustCard"
* Object Type: AreaButton
* Object Name: "Button1"
* Script Name: OnRelease
* Line Number: 5
* -> Do "Ellipse",ThisColor,65,35
*
* Card "CodeBustCard"
* Object Type: AreaButton
* Object Name: "Button2"
* Script Name: OnRelease

* Line Number: 5
* -> Do "Ellipse",ThisColor,86,35
*
* Card "CodeBustCard"
* Object Type: AreaButton
* Object Name: "Button3"
* Script Name: OnRelease
* Line Number: 5
* -> Do "Ellipse",ThisColor,107,35
*
* Card "CodeBustCard"
* Object Type: AreaButton
* Object Name: "Button4"
* Script Name: OnRelease
* Line Number: 5
* -> Do "Ellipse",ThisColor,128,35
*
* Card "CodeBustCard"
* Object Type: TextButton
* Object Name: "Compare"* Script Name: OnRelease
* Line Number: 32
* -> Do "GotItRight!"
*
* Card "CodeBustCard"
* Object Type: TextMenu
* Object Name: "Show Answer"
* Script Name: Occurred
* Line Number: 6
* -> Do "ShowCurrent"
*
*********
```

Notice that if the CanDo command for which you are searching is used in a Routine script, the format of the detail of the usage is as follows:

The name of the Routine in which the CanDo command is used

The line number inside the Routine which contains the usage
The text of the script line in which the command is used

If the CanDo command is used in a Card script (Startup, AfterStartup, or OnLeaving) or in a script associated with an Object (Occurred, OnClick, OnRelease, etc.), the format of the detail of the usage will take this alternative form:

The name of the Card on which the Card script or Object script resides
The Object type which contains the reference (may be a Card or any Object from an AreaButton to a Memo or List Object)
The name of the Card or Object containing the script which uses the CanDo command
The script name or type (AfterStartup, etc. for a Card script, OnRelease, Occurred, etc. for an Object script)
The line number inside the script which contains the command usage
The text of the script line in which the command is used

UsageOf Command {?} Unresolved will allow you to find all usages of a particular CanDo command, similarly to the **UsageOf Command {?}** command, with the added qualification that only if the first parameter to the CanDo command is an unresolved string expression will the usage be recorded in the Result Area. In this way, you can quickly find all indirect references to Cards, Routines, etc.

For example, in the CanDoMan Deck from ProPack1, certain Routines are invoked by combining a common root name with a variable value, resulting in a valid Routine name. By switching to the CanDoMan Deck by using the file requester opened by the "Deck Name" button, or by invoking the "Open Deck" menu option, we can illustrate this point. Fill in the "{?}" field with the CanDo command **Do** and select the **UsageOf Command {?} Unresolved** command. After a time, a list of all usages of the **Do** command will be placed into the Results Area. In this case, however, the only usages of the **Do** command that appear are those which involve the use of a string expression in the name of the Routine being invoked.

This command has applicable uses with every CanDo command which can accept at least one parameter, the first of which can be a string expression.

UsageOf Command {?} Resolved is the logical reverse of the previous command. It will only report the usage of commands which have a fully resolved string value as their first parameter. By selecting this command rather than the unresolved version while examining the CanDoMan Deck, will produce a completely different listing.

UsageOf Expression {?} performs an analysis within the scope of the search and will report the usage of all commands which have any parameters which include the expression named in the "{?}" field. The format of this report is identical to that of the **UsageOf Command {?}** command.

An expression in this context includes any part of an expression including logical or mathematical operators. The following are all valid parts of expressions:

(cont. next page)

+, =, -, \, |
MOUSEX
and user variable

UsageOf Function {?} will find and report on every scripting command in the Deck which contains a reference to the function named in the "{?}" field. If the word "RANDOM" were typed into the "{?}" field, every command which contained a reference to the CanDo Random function, in any parameter to that command, would be listed into the Results Area.

UsageOf String {?} is a similar command to the **UsageOf Expression {?} command**, except that it will identify script commands with parameters which contain the literal, case sensitive string specified in the "{?}" field. If the string specified is part of an expression, it will *not* be found. For example, if the "{?}" field contained the string "Fred", the following CanDo command would be found:

Do "Frederick"

whereas this command would not:

Do "Fred"||Name

because this second example is not a string literal.

UsageOf Variable {?} is a command that performs essential the same task as **UsageOf Function {?}** except that only variable names can be searched for by this method. The variable can be a user variable or a CanDo System variable, such as MOUSEX. The variable name must be entered into the "{?}" field before the command is selected.

CrossRef Menu Commands

CrossRef also has two menus, "Project" and "Output", through which certain features are available.

The "Project" menu allows you to open a Deck to analyze ("Open Deck..."), which is the same as pressing the "Deck Name" button at the far left side of the CrossRef window. You can also get information about the CrossRef tool ("About"), as well as quit the program ("Quit"). The program can also be quit by pressing the close button in the top left corner of the window.

The "Output" menu offers three controls affecting the Results Area. "Clear" will completely empty the Results Area. Be careful, there is no "undo" for this function! The "Print to..." menu function will open a file requester to assist you to choose a file to which to direct printed output, if you so choose. The last option, "Print", performs the very same function as the "Print" button directly underneath the "Deck Name" button on the left side of the CrossRef window. It will cause the Results Area to be printed.

Be a part of it.

Just what you need...

The CanDo Intro Pak

...to get up to speed!

INOVAtronics announces a comprehensive, no-nonsense guide to getting started with CanDo, the unique, powerful software authoring system for the Amiga. Consisting of a 115 page book and disk, the tutorial-intensive ***CanDo Intro Pak*** includes several CanDo example decks which complement and progress beyond the tutorial decks provided with CanDo. Each of these increasingly complex decks is accompanied by clearly written, easy to understand tutorials revealing how to make practical use of CanDo's capabilities, from the simplest to the most advanced. So, take advantage of the situation!

This package also includes complete info on:

- ***CanDo installation and set-up**
- ***Setting of Tool Types** (a very powerful feature)
- ***Valuable tips from the CanDo experts** on the following subjects: Buttons & Menus, Scripting Techniques, Technical Topics, and General Aids

Intro Pak requires CanDo v1.02. If you have a previous version of CanDo, you can upgrade by sending in your two CanDo release disks (Program and Extras) and $4.00.

CanDo Intro Pak
from INOVAtronics, Inc.
Only $39.95 plus $3.50 shipping
Visa and MC accepted. To order, call (214)340-4991 voice or (214)340-8514 FAX

CanDo.