TODO

App. E: Memory Bank Structures

The AMOS Professional package comes complete with an invaluable range of accessory programs, allowing the
serious programmer to generate all of the requirements needed to produce commercial quality products. We have
provided as much material as is possible on disc, but there is a finite limit to the disc space available. To be blunt,
we do not have the magnetic resources to cover every possibility for creative AMOS Professional programming. The
system has been designed to be infinitely accommodating, and it is our intention to expand and improve the core
system to meet all of your requirements.
After using AMOS Professional, you may well identify an area which has scope for a new accessory. Perhaps you
have a special interest in a music editor, or sound effects, or speech synthesis, graphic tweening, DTP, in fact any
aspects of enhancing the AMOS system. We are always very interested in making contact with creative, innovative
talent, that can take the AMOS Professional system to its next phase of evolution. In other words, if you feel that
you can create an important new AMOS Professional accessory, you should submit it to us on disc, along with all
relevant documentation.
But...
Before you can create such a vital accessory, you must understand the internal structure of the various AMOS
Professional memory banks. This will allow you to generate such banks directly from your AMOS Professional
programs.
Much of the information in this Appendix is very technical, and it is likely to prove heavy going for anyone who is
not an experienced programmer. In order to exploit it successfully, you are going to have to explore the Amiga's
memory very carefully. There will only be one major warning in this Appendix: if you make a mistake in the
realms of memory bank manipulation, you will crash your computer!
For those genuine professional programmers who are about to persevere with innovation and exploitation of the
AMOS Professional system, you should be able to generate some amazing accessories by analysing this information.
Go ahead. Make our day!
General Information
Each AMOS Professional program can have its own unique list of associated memory banks.
All banks are introduced by a standard memory header.
In the original AMOS package, the memory banks were represented using an internal array of just fifteen addresses.

These were used to hold the current memory location of each bank assigned to a standard AMOS program. The
evolved AMOS Professional package uses a much more flexible system, allowing as many memory banks as you
need.
Memory banks are now stored using a "linked list", which works like a chain, with each header containing a
"pointer" to the next header in the Amiga's memory. AMOS Professional is able to search through this list of
headers to find the address of any bank in memory. It starts from the top, and works downwards until it finds the
required bank number. At the end of the chain, the address is terminated with a value of zero.
14.E.01
Memory Bank Structures
As well as this superb new memory pointer, the header contains special flags which tell AMOS Professional the
type of current bank under surveillance. For mere mortals, a name is supplied in simple Ascii format as well!
Here is a list of currently allowable bank names:
Sprites
Icon
Music
Amal
Menu
Samples
Pic.Pac
Resource
Code
Tracker
Data
Work
Chip
Fast
You are more than welcome to add your own bank definitions to this list.
Memory bank headers
The header is stored in the following format:
Header dc.l Address_Of_Next_Bank * Start-24
dc.l Length_Of_Bank + 16 * Start-20
dc.l Number_Of_The_Bank * Start-16
dc.w Flags * Start-12
dc.w Free_For_Future
dc.b "Namebank" * Start-8
Start *

- Data goes here * Returned by START function
  There now follows an explanation of each of the above Header components.
  Address_Of_Next_Bank
  This is the address of the next bank in the memory chain. The list is terminated with a value of zero. Note that each
  new bank is added to the top of the list so the last bank that has been reserved will be the first bank in the chain.
  These pointers are swapped around whenever the BANK SWAP command is called from

Length_Of_Bank+16
For Sprite and Icon banks, this refers to the size of the pointer/palette list. Otherwise the length of the bank is in
bytes.
Number_Of_The_Bank
The number of the bank is held as a standard AMOS Professional integer in four bytes, but RESERVE and ERASE
will only make use of the lower two locations (Start-14). So although there can be a theoretical maximum of
2,147,483,647 banks, AMOS Professional will only manipulate banks numbered from 1 to 65535.
Please note that if you poke in a number above 65535, this bank may only be deleted with an ERASE ALL
command.
Flags
The flags are stored as individual binary bits, and have the following meaning:
Bit #0: 1 => This sets a permanent DATA Bank which will be saved with an AMOS Professional program.
0 => This is a temporary WORK Bank which will be erased by an ERASE TEMP
command, and discarded every time the program is run.
Bit #1: 1 => CHIP memory bank, used for Objects which are to be displayed on the screen, or items played through the Amiga's sound chips:
Sprites, Bobs, Icons, Samples and Music.
0 => This is a FAST memory bank.
Please note that if there is no FAST memory available, all of the FAST banks will be stored in CHIP RAM instead.
This allows you to use the same definitions on any Amiga!
Bit #2: 1 => Object bank (list of pointers)
0 => Normal, one-section bank
Bit #3: 1 => Icon bank (list of pointers)
0 => Normal, one-section bank
DATA and WORK bits can be changed as much as necessary, but you should never alter ICON, BOB, CHIP or
FAST flags. Doing so is absolutely guaranteed to crash the Amiga the next time a bank is reserved or erased.
Although you can have more than one Sprite bank in memory, only bank number 1 will be used to display Sprites
and Bobs. Similarly, only bank number 2 can be used for Icons.
If you need to use several Object banks, images can be placed into any bank you wish, and these banks can be
switched using the BANK SWAP command, before they are displayed.

For example:
X> Bank Swap 1,10: Rem Swap over banks 10 and 1, bank 10 is the new Sprite bank
Free For_Future
This bank is reserved for the future expansion of AMOS Professional. Use it at your peril!
"Namebank"

This holds the name of a bank. The name is simply a string of eight characters poked into memory, and it can be
changed or altered at will. This makes it easy to create your own bank types for home-grown accessories. Note that
only printable characters should be used, with Ascii codes greater than 32.
Start
This is the address returned by the START function, and indicates the beginning of the actual data.
WORK BANKS and DATA BANKS
These are the backbone of many AMOS Professional programs, and are used to hold a variety of types of
information. They may be stored either in memory or on disc, as follows:
Work Banks and Data Banks stored in memory
WORK BANKS
Header dc.l Next_Bank Start-24
dc.l Length_Of_Bank + 16 Start-20
dc.l Number_Of_The_Bank Start-16
dc.w Flag Start-12 (2=Chip Work or 0=Fast Work)
dc.w Free_For_Future Start-10 (Do not touch!)
dc.b "Work " Start-8
Start

- Data goes here
  ds.b Length_Of_Bank
  DATA BANKS
  Header dc.l Next_Bank Start-24
  dc.l Length_Of_Bank + 16 Start-20
  dc.l Number_Of_The Bank Start-16
  dc.w Flag Start-12 (3=Chip Data or 1=Fast Data)
  dc.w Free_For_Future Start-10 (Do not touch!)
  dc.b "Data " Start-8
  Start Returned by START function
- Data goes here
  ds.b Length_Of_Bank Returned by LENGTH function
  14.E.04
  Memory Bank Structures
  Work banks and data banks stored on disc
  Before AMOS Professional saves your banks onto disc, the header is discarded and replaced by the following:
  dc.b "AmBk"
  dc.w Number_Of_The_Bank
  dc.w Flag
  dc.l Length_Of_The_Bank + 8
  dc.b "NameBank" * 8 bytes
  ds.b Length_Of_The_Bank * the bank itself!
  These files will normally end with ".abk" and can be loaded into AMOS Professional Basic using the LOAD
  command.
  Saving several Banks at once
  AMOS Professional allows you to save a group of banks in a single ".abs" file. The format of these files is as
  follows:
  dc.b "AmBs"

dc.w Number_Of_Banks
The memory banks are then listed onto the appropriate disc, one after another.
Format of Object banks and Icon banks
As before, the manner in which Objects and Icons are stored in memory will be examined, followed by an
explanation of their storage on disc.
Object banks and Icon banks stored in memory
Icons and Objects are stored in a special way. Rather than hold the data in a single continuous package, AMOS
Professional splits these banks into a separate list of images. These images are stored in their own independent
memory locations, and are scattered through the Amiga's Chip Ram. This makes it very easy to add or delete
images, and avoids the problem of "garbage collection".
However, this does require you to take a little care when accessing images directly from your programs. Never try
to FILL or COPY data directly to the Object or Icon bank. Do not try to load or save your images with BLOAD or
BSAVE, these commands will not work. Use LOAD and SAVE instead.
The locations of the images are held in a list of pointers, which can be found immediately after the header.
In order to remain compatible with the original AMOS system, Object banks are indicated by the name "Sprites"
rather than "Objects", but they can be used to hold either Sprite or Bob images as required.
14.E.05
Memory Bank Structures
Header dc.l Address_Of_Next_Bank Start-24
dc.l Length_Of_Bank + 16 Start-20
dc.l Number_Of_The_Bank Start-16
dc.w Flag Start-12 (5=Objects or 9=Icons)
dc.w Free_For_Future Start-10 (Do not touch!)
dc.b "Sprites " Start-8 (or "Icons ")

- Start of the bank Returned by START function
  Start dc.w Number_Of_Images Returned by LENGTH function
- ".img" stands for the number of the image
- There is a separate pointer for each image in the bank
  REPT Number_Of_Images For IMG=1 To No_Of_Images
- Store pointer values
  dc.l Image_Address.img Address of Image
  dc.l Mask_Address.img Address of Mask (if defined)
  ENDR
- Colour palette (32 words). This holds the colour values used by your images
  dc.w 32
  Image_Address.img
  If you have created a blank image using INS BOB, the address of the image will be 0 (zero). In this case, there is
  obviously no mask address either.
  Mask_Address.img
  This can have different values. If the value equals zero, the mask is not yet calculated. It will be created when the
  image is assigned to the Bob automatically. If the value is -1 the user has called the NO MASK command, so

AMOS Professional will not bother with the mask. If the Mask_Address.img is greater than zero , it will hold the
address of the mask in Chip memory.
Each image has a separate data area:
Image_Address.img
dc.w X_Size (Width in words = pixel size/16)
dc.w V_Size Height in lines
dc.w Number_Of_Planes Number of planes (1 to 6)
dc.w Hot_Spot_X OR Flipping_Flags Holds X control point + extra flags
dc.w Hot_Spot_V

- Image data
  REPT Number_Of_Planes
  dcb.w X_Size * Y_Size
  ENDR
  The image definition is merely a small bitmap containing the actual picture. The planes are stored one after another,
  starling from plane 0.
  14.E.06
  Memory Bank Structures
  X_Size This holds the width of the image, divided by 16.
  Y_Size Stores the height of the image in screen lines.
  Number_Of_Planes A value from 1 to 6 which sets the number of colour planes.
  X_Hot,Y_Hot These set the position of the hot spot of the image
  Flipping_Flags This is used by the HREV, VREV and REV functions.
  The Bob flip commands were added in AMOS V1.21 and rather than redefine the entire system, Francois Lionet
  simply grabbed a couple of bits at the top of the HOT SPOT, and used them directly for the new options. The xcoordinate
  was truncated to 14 bits (signed), so you may now set HOT SPOT values between -4096 and 4096, which
  is hardly a limitation!
  Bit #15 indicates that the image has been flipped from left to right, and bit #14 informs AMOS Professional that the
  image has been turned upside down.
  If the mask has been defined, it only contains one bitplane. Bits with a value of zero are transparent, allowing the
  background to be seen through them, and bits with a value of 1 are opaque.
  Mask_Address:
  dc.l Size_Of_The_Mask In_Bytes
  dcb.w X_Size * Y_Size
  Object banks and Icon banks stored on disc
  An Object or Icon bank is stored very differently on disc, as all information relating to the pointer is discarded.
- When saving a Sprite bank the header starts with:
  dc.b "AmSp"
- If it is an Icon bank:
  dc.b "Amlc"
- The rest of the header is common to both Objects and Icons:
  dc.w Number_Of_Objects
  REPT Number_Of_Objects
  dc.w X_Size
  dc.w Y_Size

dc.w Number_Of_Planes
dc.w X_Hot_Spot
dc.w Y_Hot_Spot
REPT Number_Of_Planes

- The actual image goes here
dcb.w X_Size * Y_Size
ENDR
ENDR
- 32 colour palette holding the image colours
dcb.w 32

Memory Bank Structures
Please note the following three points:
If a Sprite or Icon is empty, AMOS Professional will only save this:
dc.w 0
dc.w 0
dc.w 0
dc.w 0
dc.w 0
The mask is not saved by AMOS Professional!
All Objects or Icons are flipped back to their original state before they are saved, so the bits 14 and 15 of the X Hot
Spot are always zero.
MUSIC BANKS
In this section, music banks held in memory will be dealt with first, followed by an examination of music banks
saved onto disc.
Music banks stored in memory
The AMOS Professional Music system is stored as an ,extension, so it is completely separate from the rest of the
AMOS Professional language. The source code is available and can be changed or modified to your own needs. This
means that the system will not be made redundant by any future developments in the world of Amiga music!
Internally, AMOS Professional Music is totally different from the standard Soundtracker format. Music is not coded
in parallel, that is to say with all notes for all of the voices in 16 bytes, but in a more efficient "track" system. This
system is also a little more complex.
Each voice has its own individual track, and the delays between each note are not fixed as in Soundtracker, but
coded in the note itself. Pauses are achieved by counting a delay value down to zero.
Labels are not stored as part of the notes, but are entered just before them, using two bytes. The advantage of this
technique is that up to 128 different labels may be employed, using a full byte for the parameter values. You can
also insert several labels one after the other, and the effect will be heard when the next note is played.
This structure makes the AMOS Professional music player very versatile. After appropriate conversion, it can play
music like Soundtracker or IFF music files.
Music banks are completely re-locatable, and are structured in three, independent, main

parts:
Instruments: this holds the sample data for each instrument in the composition.
Musics: this contains a list of pattern numbers to play in sequence.
Patterns: this a simple list of notes.
14.E.08
Memory Bank Structures
At the start of the music bank, AMOS Professional stores offsets to the various components of the music.
Header dc.l Next_Bank
dc.l Length_Of_Bank + 16
dc.l Number_Of_The_Bank
dc.w Flag
dc.w Free_For_Future
dc.b "Music " * 8 Letters
Start: dc.l Instruments_Start * Offset to first instrument
dc.l Musics_Start * Offset to first music
dc.l Patterns_Start * Offset to first pattern
dc.l 0 * Free for future!

- The Instrument part
  Instruments:
  dc.w Number_Of_Instruments
- For each instrument (.inst represents the number of the instrument)
- Repeat
  REPT Number_Of_Instruments
- Offset to sample attack part
  dc.l Attack.inst_Instruments
- Offset to instrument loop. If there is no loop, this points to a null sample at the start
  dc.l Loop.inst Instruments
- Length of the samples, in words (ready to Doke into the circuitry)
  dc.w Attack_Length.inst
  dc.w Loop_Length.inst
- Volume level
  dc.w Volume.inst
  dc.w Total_length.inst
- Name of the instrument in Ascii
  dc.b Name_Of_Instrument_In_16_Bytes
  ENDR
- Until Last instrument
- End of instrument definitions
- Now comes the null sample
  dc.w 0,0
- And the sample data for each instrument, one after another
- Repeat for every instrument
  REPT Number_Of_Instruments
  Attack.inst:
  dcb.b Sample ... *Sample data for attack
- If a loop is defined:
  Loop.inst:
  dcb.b Sample ... * Loop sample goes here
  ENDR
- Until Last Instrument
  *

14.E.09
Memory Bank Structures

- The Music part starts here, as a list of patterns to be played in sequence
  Music:
  dc.w Number_Of_Musics
- ".mus" is the number of the music ...
- Repeat for each piece of music
  REPT Number_Of_Musics
  dc.l Music.mus_Music * Offset to Pointer list
  ENDR
- End repeat
- Repeat for each bit of music
  REPT Number_Of_Musics
  Music.mus:
  dc.w Tempo
  dc.w List_patterns_voice_0 - Music.mus * Offset to Voice 0
  dc.w List_patterns_voice_1 - Music.mus * Offset to Voice 1
  dc.w List_patterns_voice_2 - Music.mus * Offset to Voice 2
  dc.w List_patterns_voice_3 - Music.mus * Offset to Voice 3
  dc.w 0 *Free for extension
- We now add the list of the pattern numbers to play for each voice
  List_patterns_voice_0:
  dc.w "" * Patterns for voice 0
  List_patterns_voice_1:
  dc.w "" * Patterns for voice 1
  List_patterns_voice_2:
  dc.w "" * Patterns for voice 2
  List_patterns_voice_3:
  dc.w "" * Patterns for voice 3
  ENDR
- End Repeat
- The last bit holds the pattern definition
- ".pat" stands for the number of the pattern
  Patterns:
  dc.w Number_Of_Patterns
- Repeat for each pattern
  REPT Number_Of_Patterns
- Offsets to the note values for each voice
- Each individual pattern can be safely assigned to ANY voice
- Simply set the offsets accordingly
  dc.w Voice_0_Note_list.pat - Patterns * Offset to voice 0 notes
  dc.w Voice_1_Note_list.pat - Patterns * Offset to voice 1 notes
  dc.w Voice_2_Note_list.pat - Patterns * Offset to voice 2 notes
  dc.w Voice_3_Note list.pat - Patterns * Offset to voice 3 notes
  ENDR
- End Repeat
- And now for the note list, one after the other ...
- Repeat for each pattern
  14.E.10
  Memory Bank Structures
  REPT Number_of_patterns
- We will now define a separate note list for each voice

- This is NOT essential, as the notes are TOTALLY independent of the voice number
- So the same note list can be used for ANY of the four voices if required
  *
  Voice_0_Note_List.pat:
  dc.w "" * All the notes for voice 0 go here
  Voice_1_Note_List.pat:
  dc.w "" * All the notes for voice 1 go here
  Voice_2_Note_List.pat:
  dc.w "" * All the notes for voice 2 go here
  Voice_3_Note_List.pat:
  dc.w "" * All the notes for voice 3 go here
  ENDR
- End Repeat
  The Patterns
  Unlike the Soundtracker system, Patterns are held as a simple list of notes, and they can be assigned to any of the
  four voices independently. Providing that the correct offset values are set, you can play the same pattern through all
  of the available voices simultaneously.
  The AMOS Professional music format is closer to IFF music format than the standard Soundtracker system. Each
  effect, every instrument and each note is defined by a specific label. Several labels can be inserted in a sequence,
  and the AMOS Professional music routines will execute them one by one, until it finds the actual note to be played
  through a loudspeaker.
  The labels are stored as two-byte words, using the following system:

- A normal note:
  dc.w %0000ppppppppppp

- ppppppppppp defines the "period" of the sample
- This will be poked directly into the Amiga's sound chips
- Please see your technical reference manual for more details
  The note will be played immediately, using the current instrument assigned to the voice.
  Labels are defined by setting bit 15 of the note to 1. The general format is as follows:
  dc.w %11111111 pppppppp
- 11111111: the number of the label
- pppppppp: a parameter value
  Here is a full list of the possible label types:

- PATTERN_END label 0
  dc.w %10000000 00000000
- SET_VOLUME label 3(1 and 2 are presently unused)
  dc.w %10000011 vvvvvvvv

- vvvvvvvv : volume level from 0 to 63
  14.E.11
  Memory Bank Structures

- STOP_EFFECT label 4
  dc.w %10000100 00000000
- REPEAT label 5
  dc.w %10000101 rrrrrrrr

- rrrrrrrr : number of times to repeat
- LED_ON label 6
  dc.w %10000110 00000000
- LED_OFF label 7
  dc.w %10000111 00000000
- SET_TEMPO label 8
  dc.w %10001000 tttttttt
- tttttttt : new tempo from 0 to 63
- SET_INSTRUMENT label 9
  dc.w %10001001 11111111
- 11111111 : number of the new instrument
- SET_ARPEGGIO label 10
  dc.w %10001010 aaaaaaaa
- aaaaaaaa : value of the arpeggio
- SET_PORTAMENTO label 11
  dc.w %10001011 pppppppp
- pppppppp : value of the portamento
- SET_VIBRATO label 12
  dc.w %10001100 vvvvvvvv
- vvvvvvvv : value of the vibrato
- SET_VOLUME_SLIDE label 13
  dc.w %10001101 ssssdddd
- ssss : step size
- dddd : duration
- SLIDE_UP label 14
  dc.w %10001110 ssssssss
- ssssssss : frequency shift
- SLIDE_DOWN label 15
  dc.w %10001111 ssssssss
- ssssssss : frequency shift
- DELAY label 16
  dc.w %10010000 dddddddd
- dddddddd : delay duration in 1/50th of a second
  This label is normally used right after a note definition to pause for a moment while the note is played
- JUMP label 17
  dc.w %10010001 pppppppp
- pppppppp : the number of a pattern you want to jump to
  Please note the following comments:
  Everything is relocatable.
  14.E.12

Memory Bank Structures
The AMOS Professional Music player does not modify anything in the bank before the music is played, unlike
Soundtracker.
The number of instruments is virtually unlimited, with a choice of 65536!
There is an unlimited number of patterns.
With a little work, you are able to save a lot of space. The same pattern can be re-used by different songs, and
may be repeated several times in your Soundtrack.
Music banks stored on disc
The AMOS Professional music banks are saved to disc "as is", with only a simple header.
dc.b "AmBk"
dc.w Number_Of_The_Bank
dc.l $80000000 + Length_Of_The_Bank
Note that $80000000 indicates a CHIP memory bank.
SAMPLE BANKS
All sample banks are loaded in CHIP Ram.
dc.b "Samples" Start-8 Name of the bank
Start dc.w Number_Of_Samples

- First we store a list of pointers to the samples in memory
- These are held as offsets from the start of the bank

REPT Number_Of_Samples
dc.l Sample_XX-start XX = number of the sample
ENDR

- Now we store the samples one after the other
- Repeat for each sample

REPT Number_Of_Samples
Sample XX dc.b "Namesamp" Name of the sample in 8 bytes
dc.w Sampling_Frequency In Hertz
dc.l Sample_Length In WORDS (real length/2)
dcb.b ... samples ... The actual sample data
ENDR
On disc, the sample bank is saved directly in the above format. The disc header is exactly the same as for a CHIP
DATA bank.
AMAL BANKS
An AMAL bank can hold two separate types of information. Either a list of AMAL command strings, or a recorded
series of Object movements for use with the PLay instruction. The bank is therefore divided into sections, as shown
below:
The header
dc.b "AMAL " Start-8 Bank name, 8 bytes, Ascii
Start dc.l Strings-Start Offset to the first command string in memory
The movement table

- We start with a list of the movement table used by the PLay command
- (NN= number of the move)

- For NN=1 To the Number of Recordings

Moves dc.w Number_Of_Movements

- Pointers to the list of X coordinates
  REPT Number_Of_Movements
  dc.w (XMove_NN-Moves)/2 Offset to the X coordinates /2
  Or zero if they are not defined
  ENDR
- Location of the Y coordinates
  REPT Number_Of_Movements
  dc.w (YMove_NN-Moves)/2 Offset to Y coordinates /2
  Or zero if they are not defined
  ENDR
- Stores an eight byte name for each movement table
  REPT Number_Of_Movements
  dc.b "MoveName" 8 Bytes per move
  ENDR
- Finally here are the movement definitions themselves
  *
  REPT Number_Of_Movements
  XMove_NN dc.w Speed Recording speed in 1/50 sec
  dc.w Length_Of_X_Move Length of table in Bytes
  dcb.b ... XMove definition ...
  YMove_NN
  dcb.b ... YMove definition ...
  ENDR
  The movements are stored in the following way. The movement table uses the same format for both X and Y
  coordinates. It begins and ends with a value of zero, which terminates the list equally well if the movement is being
  played forwards or backwards.
  %00000000 End of the move
  %0ddddddd ddddddd holds the distance to be moved in pixels,
  signed on 7 bits (-128 to +128)
  This distance will be added to the current object
  coordinate to get the new screen position
  %1wwwwwww specifies the number of
  1/50 counts to wait until the next
  movement
  The AMAL programs
  AMAL command strings are stored in normal Ascii format.
  Progs dc.w Number_Of_Programs Holds the number of AMAL programs
- Offset list
  REPT Number_Of_Programs
  dc.w (Prog_NN-Progs)/2 Distance to the NN'th program
  14.E.14
  Memory Bank Structures
  measured in WORDS
  ENDR
- Programs
  REPT Number_Of_Programs
  Prog_NN dc.w Length Of Prog_NN
  dc.b "The program in plain Ascii"
  ENDR
  THE RESOURCE BANK

The Resource bank is used to hold all the control buttons and icons used by the AMOS Professional INTERFACE
commands. The Resource bank is split into three main sections. There is one area for the button definitions, another
for the command strings and a third for messages.
dc.b "Resource"
Start dc.l Images-Start * Offset to the compressed images (optional)
dc.l Texts-Start * Offset to the message list (optional)
dc.l DBL-Start * Offset to the Interface program (optional)
dc.l 0 * Reserved for future expansion

- The compressed images go here
- These are used by the UNpack, Line and BOx commands from the Interface

Images dc.w Number_Of_Images Holds the number of parts
REPT Number_Of_Images
dc.l Image_NN-Images Offset to the start of each part
ENDR

- We now enter full details of the screen from which the images were grabbed

dc.w Number_Of_Colours
dc.w Graphic_Mode In the same format as SCREEN OPEN
(Lowres, Hires, Laced)
ds.w 32 Holds the colour palette for the images
dc.w Length_Of_Name Now the name of the source image
dc.b "Full_Path_Name" This is a name in simple Ascii format
dc.b 0 Pad out the byte, if not even

- Each image is a normal packed bitmap, in "pic.pac" format
- At this moment, there are only two possible image types
- Simple image

Image_NN:
dc.b Packed_data Internal to the screen packer!

- Alternatively, the data can be a BOx definition, a Line definition
- or comments on a specific image, entered in the resource bank_maker.
- in this case, a magic number, =$ABCD will be immediately BEFORE
- the graphic data.

Res_NN:
dc.b "name " 8 bytes, Ascii
dc.w Number_Of_Images A BOx needs 9 images, Lines need 3
and a simple image has only 1
dc.w $ABCD
dcb.b Packed_Data

- These types can be mixed in any order, so it is acceptable
- to put the comment line BEFORE the button definition

14.E.15
Memory Bank Structures
*
Texts:

- This is just a simple list of strings
- Each string can hold up to 255 characters, and it is terminated by a zero
- The length has been added at the start, to make it compatible with AMOS strings.
- Each string is referred to by its number, from an Interface program.

REPT Number_Of_Strings
dc.b 0
dc.b Length

dc.b "The string in plain Ascii"
ENDR
dc.b 0

- Holds one or more Interface command strings in standard Ascii format
- Offset list
  DBL: dc.w Number_Of Programs
  REPT Number_Of_Programs Each program has own offset value
  dc.l Prog_N - DBL Offset to the Interface program
  ENDR
- Repeat for each program
  REPT Number_Of_Programs
  Prog_N: One of these for each program
  dc.w Prog_N_End - Prog_N Length of Interface string in bytes
  dc.b "The text of the program, in Ascii"
  dc.b "with a ZERO at the end..."
  dc.b 0
  Prog_N_End:
  ENDR
  COMPRESSED PICTURES (PIC.PAC)
  The internal structure of these pictures is very complex, and this explanation is limited to the header file. A full
  source listing of the compaction code is available from the extensions folder.
  The packing process makes several attempts to provide the optimum compression ratio. It packs the picture into
  small blocks which are several lines high and exactly one byte wide. The height of the blocks is continually adjusted
  until the packer finds the most suitable value for the current data.
  There are two possible cases. Either a compressed bitmap created with the PACK command, or a packed screen
  created with the SPACK instruction. The compressed bitmap is examined first.
- Magic number for a packed bitmap (Happy Birthday Francois Lionet!)
  Pkcode dc.l $06071963
- (Original X coordinate of the bitmap)/8 (in bytes)
  Pkdx dc.w x
- Y coordinate of the original source data

  Pkdy dc.w y
- Width of the bitmap in bytes (number of pixels/8)
  Pktx dc.w width/8
- Height of the bitmap in blocks
  Pkty dc.w height_in_y
- Height of each individual packing block
  Pktcar dc.w height_in_lines
- The total height of the picture can be found by multiplying Pkty by Pktcar
- Number of colour planes
  Pkplan dc.w planes
- Pointer to next data list
  PkDatas2 dc.l next_data
- Pointer to next data pointer
  PkPoint2 dc.l next_pointer

- the packed data goes here!
  Finally, a packed screen created with the SPACK instruction is examined. This is identical to the previous version,
  except for some extra information that comes before the header, as follows:
  PsCode dc.l $12031990 Code for a packed screen
  PsTx dc.w Width Width of the screen
  PsTy dc.w Height Height of the screen
  PsAWx dc.w Hard_X X coordinate of screen in hardware format
  PsAWy dc.w Hard_Y Vertical position of screen
  PsAWTx dc.w Display_Width Width of screen to area to be displayed
  PsAWTy dc.w Display_Height Display Height (set by SCREEN DISPLAY)
  PsAVx dc.w X_Offset As set by SCREEN OFFSET
  PsAVy dc.w Y_Offset Coordinate of first line to be displayed
  PsCon0 dc.w mode BPLCON0
  PsNbCol dc.w cols Number of colours
  PsNPlan dc.w planes Number of bitplanes
  PsPal dcb.w 32 Holds the colour palette