

Editor Manual

COLLABORATORS

	TITLE : Editor Manual		
ACTION	NAME	DATE	SIGNATURE
WRITTEN BY		August 9, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Editor Manual	1
1.1	Contents	1
1.2	Introduction to ChunKit	1
1.3	Amiga Rezedit	2
1.4	ChunKit Features	3
1.5	Conditions of Use	4
1.6	Using ChunKit	4
1.7	The Format Window	6
1.8	Editors	8
1.9	Executable files	8
1.10	Iff Files	12
1.11	Binary Editor	13
1.12	Using the Clipboard	14
1.13	Find	15
1.14	Bugs glitches and other snafus	17
1.15	Things to do	18
1.16	Not So Legal Junk	19
1.17	Not So Legal Junk	19
1.18	Easter Egg	19
1.19	About the Author	20

Chapter 1

Editor Manual

1.1 Contents

ChunKit Manual V1.0

- Introduction
- Features
- Conditions of Use

- Using ChunKit
 - The Project Menu
 - Preferences

- The Format Window
 - Editing Executable Files
 - Editing IFF Files

- Editors
 - Binary Editor

- Using the clipboard
- Bugs Glitches and Other Inconsistencies
- Things To Do

- Disclaimer (Not so Legal Junk)
- About the Author

1.2 Introduction to ChunKit

Computers store all data as sequential strings of numbers in one format or another. Many utilities are available to view and edit files in their most primitive state, as sequential binary data. However, since most files are structured in a more complex fashion, it is often useful to view them in such a structured format. ChunKit is a hex editor designed to view and edit iff and executable files in such a structured manner.

Rather than dumping the unformatted contents of the file to the screen ChunKit first checks to see if the file is in executable or iff format.

If so, it splits the file into it's component chunks, allowing you to examine the components of the file without having to wade through all of it's structural information. You can then add, edit or delete any part of the file.

ChunKit was born while I was experimenting with another project, Rezedit. I was trying to design a very simple iff file and was experimenting with a reader program. Since I was just testing out the datatype I had a reader program, but no writer, and thus had to edit the files by hand. The first problem I encountered was one that has always peeved me with binary editors: I couldn't create a new file. I could edit an existing file, but starting from scratch was impossible. This annoyance was minor compared to having to constantly calculate iff form sizes. After making three mistakes on a 56 byte file I went looking for an iff editor. At the time the only programs I could find wouldn't allow editing the files and crashed my machine frequently. There have been several introduced since that fulfilled some of my initial requirements, so I added features to ChunKit to parse binary and executable files.

1.3 Amiga Rezedit

The Macintosh uses an elegant system for managing it's program's data, called resources. Resources are essentially data structures that are defined externally as well as internally so that they can be edited without having to recompile the file. This has many advantages for both the programmer and the user. For example, once the programmer has set up the calls to access the resource from a file (s)he can modify the resource (the position of a gadget perhaps) without having to recompile the file, speeding the development process. On the other hand, an advanced user can change hard coded values such as strings in a foreign language, gadget positions, or even resource id's (such as device interfaces) that become modifiable under new versions of the operating system. While resource editing is an advanced operation it can often allow users to make more effective use of their software.

Because of the way the Amiga operating system is structured it would be rather easy to add resources in a useful way. The Amiga iff format is very similar to the Macintosh resource format (in fact the iff committee based the 4 character iff ID's on the Macintosh resource ID's). By linking an iff file into the programs executable and accessing it at load time the resources could be extracted. The Amiga's object oriented api would respond very well to such an addition. Almost all resources could be represented as taglist's. A GetRsrcTag() function could obtain a pointer to this taglist and pass it to functions such as OpenWindowTags(). (This could all be encapsulated in a function such as OpenWindowRsrc().) The user could open the file and open the resource hunk and the WIN resource to edit the windows TagList. As further tags were added in new releases of the operating system programs that could make use of them could be easily modified. (I.e.: adding SA_PLANAR to a screen taglist could speed up programs that could handle such a modification).

Since most resources would be implemented as taglists there would be little work involved in adding resources to all system functions except for providing the various resource stubs to the tag array functions. Most of the work involved would be in designing the resource structure, and

resolving pointers to the resources containing the appropriate data. (The easiest way to do this would seem to be to add a reloc32 block to the resource hunk that would be modified by the rezedit program as the user edited resource's.)

The resource structure itself would be one of the most important parts. One of the big drawbacks of the Mac implementation is that it shows it's age (like most of the Mac api). Several resources are limited (for example the infamous 128 font limit is due to the way fonts are implemented as system resources) and many don't do all that they should (creating an interface is unnecessarily complicated unless you are developing a nearly obsolete desk accessory). While it would be easy to provide a string resource, a few chip data resources and a taglist resource and call it quits, a much more powerful system could be achieved by having nested resources allowing for more complex functions. For example, opening a window that contained a series of gadget resources should open the window and all of the gadgets.

Since ChunKit was written while I was playing around with this idea it is basically the rezedit program that would be needed. It can edit the CODE resources (the Amiga hunks) and with slight modification could parse a data hunk as an iff file and allow editing of the various taglists within. It's modular nature would even allow a full featured GUI designer to be started when the appropriate chunks were edited. Unfortunately, I currently am not considering working further on the rezedit project. If you have any comments or feedback on the idea please send me email. If a hundred people were interested in the idea I might start to work on it, but I think this is unlikely and there are other projects I'd like to pursue that could prove more useful (and interesting).

1.4 ChunKit Features

ChunKit has the following features:

- Opens all files
 - Displays all iff files in a tree format allowing chunks to be deleted, renamed or edited. This means you can read and modify iff files easily without having to worry about iff structure.
 - Executable, object and library files can be examined and edited in a tree format. Each Hunk can be opened and it's symbol, reloc or code blocks can be examined and modified. Symbol and debug blocks can be removed to reduce size and load time.
 - New iff and binary files can be created from scratch. If you need to create a specific file there is no need to edit an existing file.
 - OS friendly, non modal gadtools interface, streamlined to be as user friendly as possible (i.e.: There are a lot of features, not a lot of buttons. The emphasis has been placed upon giving the user as many useful features as possible without an overwhelming array of options. If the program can determine something on it's own it does so.)
 - Fully supports the Amiga clip board.
 - Modular internal design so new types of files and chunks can be edited in the future.
 - Enforcer clean.
-

- Non features such as non standard file requesters have not been included. If the user prefers some other file requester they can patch it into their system and all of their applications can use it and share a common interface.

1.5 Conditions of Use

Conditions of Use

Well, I'm not going to tell you not to send me money, but I'm also not going to require you to send me any if you use ChunKit. I spent a lot of time working on ChunKit, but I also learned a lot and even occasionally enjoyed myself :-). If you really like ChunKit you could give me something I would prefer to money - a job. I'll be looking for a full time position between September and December 95 and would love to do development work on the Amiga or UNIX. I would also be interested in a position that would allow me to learn and develop for OS2 or the Macintosh.

If you do find ChunKit useful, or just interesting I would appreciate a letter, postcard, what have you, with any feedback and a description of what use you've found for ChunKit (the more unusual the better). Again I won't require you to send me a letter (I'd hate to have people lying awake at night thinking "oh oh... I forget to send Pat a letter telling him what I used his program for..." :-)) but I would really be interested in any feedback.

The only restriction I place on the use of ChunKit is that you don't use it to crack or copy other people's software. You may examine file structure and modify files for your own use where permitted, but please don't steal other people's programs or ideas.

You can distribute ChunKit freely but you cannot charge for it (above the price of the distribution medium), and you must keep the archive intact, with all files and documentation. If ChunKit is distributed on a magazine cover disk or with a commercial product I would like a letter detailing where I can obtain a copy of the magazine or product. Distributions such as the Aminet CD and Fred fish are more than welcome to include ChunKit in their distribution - I have purchased both of these archives and have found them to be invaluable.

1.6 Using ChunKit

Using ChunKit

To start ChunKit from the workbench simply click on it's icon. If you want to open a file select ChunKit and shift double click the file. From the cli type ChunKit followed by the file to open, if any.

If no file is specified the program will either prompt you for a file name or a file type (iff or binary) depending on preferences. Depending on what type of data the file contains one of two windows will open up: a format window or an editor window.

Regardless of which window opens there are several options which remain constant: the project menu and the preferences menu.

The Project Menu

The project menu is the same in all windows and contains the tools for loading, saving and creating files. All of the file I/O functions in the project menu have a common interface. All functions that delete the current file from memory (new, open and quit) will ask if you wish to save the file if it has been modified. If the current file has no file name, you will be prompted with a requester before saving. If an IO error occurs during any operation a requester appears with the a description of the error and abort and retry options. Retry will re-attempt the operation and abort will exit the operation with no further requesters. If a save operation is aborted the operation that initiated the save will also abort (i.e.: if you select new and then ask the program to save the current file before continuing, but there is an error the new operation will not delete the current file). In this case you must restart the operation and select no when the save request appears in order to delete the current file from memory.

New: Opens a new empty file for editing and deletes the file currently in memory.

Open: Prompts you for a file name with a requester and loads it into
(AO) memory. If the load fails the old file is not deleted from memory unless there was not enough memory. In this case the program frees the old file first to see if the new file can be opened with more memory. If this occurs and the save still fails the program will open an empty file to replace the one that was deleted.

Save: Saves the current file.
(AS)

Save as: Prompts you for a new file name with a requester and saves the
(AA) current file with this name. If the file already exists you will be asked if you want to overwrite it.

About: Displays a window containing the program's name, version and the authors name and email address.

Quit: Closes all windows and exits the program. If the final save incurs an error and you abort, the file will be re-opened so you can edit it to enable it to be saved (this shouldn't be necessary). If the file cannot be saved you will have to quit again and select no from the "Save" requester. Closing all windows has the same effect as selecting quit.

Preferences

Preferences are the default settings that control the behavior of ChunKit. They are stored in the programs icon and most can be modified by the Preferences menu. Both the format window and the editor windows use a subset of the full preferences menu. The available settings in the preferences menu are:

Default Bin: If this item is checked, all files are loaded as binary files even if they could be parsed as iff's or executables.

No Container: This prevents the container chunk on the clipboard from being pasted. See using the clipboard for more details.

Chunk Style: This setting only applies to iff files. Every iff chunk has a type and an id. For container chunks (FORM, LIST, CAT, and PROP) the type describes the chunk contents. For data chunks the type is simply their parent chunk's type. The convention used in commodore documentation is to display the type of data chunks. This information is largely redundant however, and not displaying it provides a visual cue as to which chunks are openable. Thus by setting style to "No ID" the type of data chunks will not be displayed. Setting it to "Standard" will display the chunks in the usual way.

Insert: This setting only applies to editors. If checked every character typed will shift all other characters to the right. Otherwise, new text will overwrite the current character. Turning on insert is currently very slow, particularly for large files and windows.

(AI)

Copy FTXT: This setting only applies to editors. If checked data copied to the clipboard will be copied as FTXT so that it can be pasted into a text editor. See using the clipboard.

Save Prefs: Selecting "Save Prefs" will save the current preferences to the program's icon. If the correct tooltypes do not exist they will be created.

Tooltypes

The following tooltypes in the program's icon control the various preference settings:

CHUNKSTYLE=NO_ID : This tooltype is identical to the "Chunk Style" preference.
 =STANDARD

INSERT =FALSE : This tooltype is identical to the "Insert" preference.
 =TRUE

COPYFTXT =TRUE : This tooltype is identical to the "Copy FTXT" preference.
 =FALSE

NOCONTAINER=FALSE : This tooltype is identical to the "No Container" preference.
 =TRUE

OPENFILEREQ : If this tooltype is present the program will open a file requester when started without a file name, instead of opening an empty chunk. It has no menu equivalent.

1.7 The Format Window

The Format Window

Structural data pertaining to file format (i.e.: iff chunks and executable hunks) is shown in the format windows. Multiple format windows can be open at once to view different parts of the same file. The format windows consist of a listview and a palette of gadgets allowing you to modify the displayed data. The window title displays the type of chunk being edited (i.e.: an iff FORM or an executable Code Hunk) and listview contains the contents of the chunk being viewed. Depending on whether or not the file being viewed is executable or iff the format window will behave differently. The currently selected chunk is displayed in the string gadget below the

listview. The type and id of iff chunks can be edited by clicking on the appropriate value in the string gadget. The new type and id will be checked for validity and if necessary modifications will be made or the operation will be cancelled. See Editing Iff Files for more details. Executable hunk and block names cannot be edited, but certain blocks that represent strings and longwords are displayed directly in the format window and can be edited in the string gadget.

The format window has the following gadgets. Gadgets that cannot be used in a given context are ghosted.

- Open: This will display the contents of the currently selected chunk. If the chunk contains binary data the appropriate editor will be opened. Double clicking on an item in the listview has the same effect as the open gadget.
- Close: This will close the currently displayed chunk and display it's parent. If the currently displayed chunk is the root chunk the window will be closed.
- Window: This will display the contents of the currently selected chunk in another format window. If the chunk contains binary data the appropriate editor will be opened.
- New: This will create a new chunk in the current window, located after the currently selected chunk. The chunk will have an ID of NONE.
- Delete: This will delete the currently selected chunk. The root chunk cannot be deleted.
- Clone: This will open another format window identical to the current one.

The format window has the following menus:

- Project : See The Project Menu
- Edit : The edit menu contains functions for moving chunks to and from the clipboard. See Using the Clipboard for more details on the way the clipboard is handled.
- Cut :This will copy the currently selected chunk to the clipboard and delete it from the file.
- Copy :This will copy the currently selected chunk to the clipboard.
- Paste:This will paste the contents of the clipboard next to the currently selected chunk. The chunk will be checked for iff or executable consistency before it is inserted in it's new location and may be modified, or the operation may be impossible. See the sections on Editing Executable Files and Editing Iff Files for more details.

Navigation: The navigation menu contains options for moving through the various chunks.

- Open (A) : This menu item has the same effect as the "Open" button.
- Close (A) : This menu item has the same effect as the "Close" button.
- Close Editors: This menu item will close all editor windows that are currently open.

Preferences:See Preferences

1.8 Editors

Editors

The actual file contents are displayed by editors. Editors are separate subprograms that can edit or view different elements of the file. ChunKit will determine what type of data is selected and launch the appropriate editor. Currently the only supported editor is the binary editor. The binary editor can open any type of data and display it in both hexadecimal and ASCII format.

1.9 Executable files

Editing Executable Files

Amiga executable files are divided into several segments called hunks. The hunks are composed of smaller segments called blocks, including such things as data, program code, debugging information and relocation tables. The only official documentation I have found on executable format is in the AmigaDos Manual and it is wildly inaccurate, or at least highly suspect in most places. Since the structured format used by ChunKit presents new names and concepts I've included a rough outline describing what all of the different components are and how they're represented in the format window.

Currently executable files are not checked as rigorously as iff files, and it is possible to create a corrupt file that cannot be reloaded if certain conventions are not followed. These are listed below. It is also not possible to create a new, empty block within an executable file. It is, however, possible to copy and paste within an executable, but it is not possible to paste a block or hunk into a place where it does not belong. For example, you cannot paste a code block into an EXEF block or a HEAD block. CODE can only exist in a HCOD. If you turn on the "No container" preference you can paste CODE blocks into all hunk types, but not into an EXEF or HEAD block (see "using the clipboard" for more details.)

Each hunk or block is represented by a four letter identifier (like a Macintosh resource) representing it's type. The current identifiers are:

EXEF:

This is the type of each executable file. Every executable file will have an EXEF as it's root node.

HEAD: This is the first block contained in all executable files. It is not contained in a hunk and contains information for the loader. It will either be the first block in a file or the first block in an overlay node (OVRN). It must contain block SIZE, FRST, LAST and TABL, and may contain one or more NAME blocks.

NAME: NAME blocks may be contained in HEAD blocks. If included the loader will attempt to open the library name included in the NAME block when the program is loaded. To edit NAME click in the string gadget. Names must be less than 256

characters and will be padded with zeros to a longword aligned length. The last name in the list will automatically be followed by a null.

SIZE: Each HEAD block must contain exactly one SIZE block. It must follow any NAME blocks. The size will be the table size of the TABL block (the size of the TABL divided by four). If this value is incorrect it will not be possible to reload the file as an executable. To edit SIZE click in the string gadget.

FRST: Each HEAD block must contain exactly one FRST block. It must follow the SIZE block. This is the number of the first hunk that the loader will load into memory. (Hunks are numbered starting with zero). To edit FRST click in the string gadget.

LAST: Each HEAD block must contain exactly one LAST block. It must follow the FRST block. This is the number of the last hunk that the loader will load into memory. (Hunks are numbered starting with zero). To edit LAST click in the string gadget.

TABL: Each HEAD block must contain exactly one TABL block. It must follow the LAST block. The table contains information for keeping track of the hunks it loads into memory.

HUNK: Hunks are the building blocks of executable files. When a program is run the hunks are loaded into memory by the loader. Each hunk gets it's own contiguous chunk of memory. Thus fewer chunks are more efficient, but smaller can fit into smaller memory areas. since each hunk should contain a block of relocatable data they are named according to the type of relocatable data. If there is no data (I don't even know if this is legal) they are simply called HUNK. If they contain a code block they are type HCOD. If they contain DATA they are type HDAT, and if they contain a bss block they are type HBSS. Hunks may also contain other blocks with debugging or relocation information.

UNIT: Hunk's may contain one UNIT block. Only object files should contain UNIT blocks, as they indicate the start of a program unit. To edit UNIT click in the listview.

NAME: Hunk's may contain one NAME block. Only object files should contain NAME blocks, as they are used to merge hunks together. If you want to force two hunks to be merged by the linker give them the same NAME. If you want to prevent two hunks from being merged give them different NAME's. To edit NAME click in the listview.

CODE: Hunk's may contain one CODE block. CODE blocks contain assembly language instructions. They are one of the three relocatable block types and may be loaded into chip, fast or any memory depending on the last character of their name. A third type of memory, extended is not currently supported by the loader and thus is not supported by ChunKit. CODE blocks must have length that is divisible by four. If a CODE block is made that is not longword aligned and the file is saved it will not be possible to load the file as an executable.

DATA: Hunk's may contain one DATA block. DATA blocks contain binary data that will be available to the program. They are one of the three relocatable block types and may be loaded into chip, fast or any memory depending on the last character of their name. A third type of memory, extended is not currently supported by the loader and thus is not supported by ChunKit. DATA blocks must have length that is divisible by four. If a DATA block is made that is not longword aligned and the file is

saved it will not be possible to load the file as an executable.

BSS : Hunk's may contain one BSS block. BSS blocks describe the size of a chunk of memory that the loader will allocate for the program's use. This memory will be cleared to all zeros. They are one of the three relocatable block types and may be allocated in chip, fast or any memory depending on the last character of their name. A third type of memory, extended is not currently supported by the loader and thus is not supported by ChunKit.

RE32: Hunk's may contain one RE32 block. RE32 blocks contain relocation information pertaining to the current hunk. RE32 blocks contain one or more pairs of HNKN and RELO sub blocks. The HNKN and RELO sub blocks in an RE32 block contain 32 bit offsets to be added to 32 bit objects.

RS32: Hunk's may contain one RS32 block. RS32 blocks are identical to RE32 blocks except that the HNKN and RELO sub blocks in an RS32 block contain 16 bit offsets to be added to 32 bit objects.

RE16: Hunk's may contain one RE16 block. RE16 blocks are identical to RE32 blocks except that the HNKN and RELO sub blocks in a RE16 block contain 16 bit offsets to be added to 16 bit PC relative offsets in code. RE16 hunk's only appear in object files as the hunks to which they refer are merged into the current hunk when the program is linked.

REL8: Hunk's may contain one REL8 block. REL8 blocks are identical to RE32 blocks except that the HNKN and RELO sub blocks in a REL8 block contain 8 bit offsets to be added to 8 bit PC relative offsets in code. REL8 hunk's only appear in object files as the hunks to which they refer are merged into the current hunk when the program is linked.

DR32: Hunk's may contain one DR32 block. DR32 blocks are identical to RE32 blocks except that the HNKN and RELO sub blocks in a DR32 block contain 32 bit offsets to be added to 32 bit objects. The offsets referred to in the RELO sub blocks are the distance from the base of the current block to the base of the hunk specified in the HNKN sub block. DR32 hunk's only appear in object files as the hunks to which they refer are merged into the current hunk when the program is linked.

DR16: Hunk's may contain one DR16 block. DR16 blocks are identical to RE32 blocks except that the HNKN and RELO sub blocks in a DR16 block contain 16 bit offsets to be added to 16 bit objects. The offsets referred to in the RELO sub blocks are the distance from the base of the current block to the base of the hunk specified in the HNKN sub block. DR32 hunk's only appear in object files as the hunks to which they refer are merged into the current hunk when the program is linked.

DRE8: Hunk's may contain one DRE8 block. DRE8 blocks are identical to RE32 blocks except that the HNKN and RELO sub blocks in a DRE8 block contain 8 bit offsets to be added to 8 bit objects. The offsets referred to in the RELO sub blocks are the distance from the base of the current block to the base of the hunk specified in the HNKN sub block. DR32 hunk's only appear in object files as the hunks to which they refer are merged into the current hunk when the program is linked.

HNKN: Each of the different type of relocatable blocks contain one or more HNKN block's. The HNKN is the number of

the hunk to be used by the next RELO block.

RELO: Each HNKN block must be followed by a RELO block. The RELO block contains a series of offsets into the current hunk's relocatable block. The values at these addresses will have the address that hunk number HNKN was loaded into added to them. Thus, references in the current hunk to the specified external hunk will point to the correct address. RELO blocks must have length that is divisible by four. If a RELO block is made that is not longword aligned and the file is saved it will not be possible to load the file as an executable.

EXTR: Hunk's may contain one EXTR block. Extr blocks contain a list of DEF, ABS, RES, REF3, REF1, REF8, DRF3, DRF1, DRF8 and COMM sub block's. EXTR block's should not appear in executable files as the references should be resolved by the linker.

DEF : EXTR block's may contain one or more DEF sub blocks. Each DEF contains one NAME and one SMBO block. The NAME is the name of the symbol, and SMBO is the offset of the symbol in the relocatable block.

ABS : EXTR block's may contain one or more ABS sub blocks. Each ABS contains one NAME and one SMBO block. The NAME is the name of the symbol, and I'm not quite sure what the SMBO is. (Sorry).

RES : EXTR block's may contain one or more RES sub blocks. Each RES contains one NAME and one SMBO block. The NAME is the name of the library?, and I'm not quite sure what the SMBO is. (Sorry).

REF3: EXTR block's may contain one or more REF3 sub blocks.

REF1 Each REF3 contains one NAME and one SREF sub block. The

REF8 SREF block contains a list of symbolic references

DRF3 within the current block. If it is within a REF3 sub

DRF1 block the SREF contains 32 bit references. Within a REF1

DRF8 sub block 16 bit references, and 8 bit references in a REF8 sub block. Finally DRF3, DRF1 and DRF8 sub blocks contain base relative references.

COMM: EXTR block's may contain one or more COMM sub blocks. Each COMM contains one NAME, one COMM and one SREF sub block.

SMBO: Hunk's may contain one SMBO block. SMBO blocks contain pairs of NAME and SYMB sub blocks. SMBO block's contain information for symbolic debugging of a file and can thus be deleted from executable files where the programmer has forgotten to do so to reduce size and optimize loading. NOTE: The actual format of SMBO blocks is identical to DEF ABS or RES sub blocks contained within an EXTR block, however since SMBO blocks only contain SYMB sub blocks the sub blocks are omitted for clarity. What this means is that opening a SMBO block reveals a list of NAME and SYMB blocks, whereas an EXTR will have an extra level of DEF ABS or RES blocks before the NAME and SYMB blocks can be seen.

DEBU: Hunk's may contain one or more DEBU block's. DEBU blocks contain raw debugging information, that is compiler specific. Debug blocks may be deleted from an executable to reduce size and optimize loading. DEBU blocks must have length that is divisible by four. If a DEBU block is made that is not longword aligned and the file is saved it will not be possible to load the file as an executable.

OVER: If a file uses overlays for it's loading there will be an OVER block after the last hunk loaded initially (specified in the HEAD). These hunks will typically contain the overlay manager code. The OVER block contains SIZE, MAXL, OTABL and TABL sub blocks.

SIZE: Each OVER block must contain exactly one SIZE sub block. The size will be the upper bound of the complete overlay TABL (the size of the TABL divided by four). If this value is incorrect it will not be possible to reload the file as an executable. To edit SIZE click in the string gadget.

MAXL: Each OVER block must contain exactly one MAXL sub block after the SIZE block. MAXL is the maximum level the overlay table uses (the first level is zero). If this value is incorrect it will not be possible to reload the file as an executable. To edit MAXL click in the string gadget.

OTABL: Each OVER block must contain exactly one OTABL sub block after the MAXL block. The OTABL is the ordinate section of the overlay table and should contain MAXL + 1 longwords all equal to zero (kinda redundant eh?).

TABL: Each OVER block must contain exactly one TABL sub block after the MAXL block. The TABL is the actual overlay table, with $24 * (\text{MAXL} - 1)$ entries.

OVRN: If a file uses overlays there will be a series of overlay nodes (OVRN) following the overlay table (OVER). Each OVRN contains a HEAD block followed by the hunks that will be loaded with that overlay (either HUNK, HCOD, HDAT or HBSS).

LIB : The new format link libraries contain a LIB block. The LIB actually contains all of the hunks of the library. LIB blocks can only appear in executable files.

INDX: LIB block's are usually followed by and INDX. The INDX blocks have a rather complex structure, but currently ChunKit does not parse them and merely opens them as a large binary block. INDX blocks must have length that is divisible by four. If an INDX block is made that is not longword aligned and the file is saved it will not be possible to load the file as an executable.

1.10 Iff Files

Iff Files

Interchange file format was developed early in 1985 by electronic arts to provide a flexible format for exchanging data between applications. It consists of a syntax for encapsulating data and a series of file types that fit within this syntax. There is a very good explanation of the iff format in appendix A of "The Amiga Rom Kernel Manual: Devices". For the purposes of editing iff files I have included a brief summary of iff format, and the conventions that ChunKit applies to them.

Iff files are built out of chunks. Every chunk is identified by a four character ID. There are two main chunk categories: container chunks and data chunks. Container chunks contain other chunks, and data chunks contain binary data. By nesting different data chunks within a single container, applications can define different attributes in an upwardly compatible way as well as providing different representations of the same data for different uses.

The four container chunks are FORM, LIST, CAT and PROP. Each one has a four character type identifier describing the data contained within. Every iff file must have a FORM, LIST or CAT chunk as it's root chunk. The type name for container chunk's is more restricted than for other chunks in order to make it possible to use the type in the file name on limited file systems such as MSDOS. If you attempt to give a container chunk an inappropriate type you will receive a warning. If you attempt to rename the root chunk such that it is no longer a container chunk ID ChunkIt will give you a warning. Furthermore, each iff file can only have one chunk at the root level. If you attempt to insert a chunk into the root level you will receive an error.

FORM chunks are by far the most common type of container chunk. FORM's can contain other chunks and even other FORM's. If you rename a FORM chunk such that it is no longer a container chunk you will be warned and asked if you wish to delete it's contents.

LIST chunks are a special type of container chunks for containing rather complex data. LIST's contain normal FORM, LIST and CAT chunks like normal FORM's but they can also contain PROP chunk's which provide global defaults for the contained chunks. LIST chunks cannot contain regular data chunks, and you will receive an error if you try to insert a data chunk into one.

PROP chunks can only reside in LIST chunk's. PROP chunks can only contain data chunks and trying to insert container chunks into a PROP will result in an error. PROP chunks must appear before any other chunks in LIST chunks and will automatically be moved to the top of the LIST. PROP chunks (and thus LIST's) are typically hard to parse because of the nested behavior they exhibit. If a LIST contains another LIST there can be multiple PROP's that override each other's settings, as well as the settings contained in the FORM, LIST or CAT chunk's after the PROP chunk's. Thus, it is usually necessary to use a stack (as provided by the iffparse library) to process the properties stored in PROP chunks correctly.

CAT chunks are similar to FORM chunks except that they contain a series of similar data (i.e.: a group of ILBM's or FTEXT chunks).

All other chunks are data chunks. They contain the actual data of the iff file. If a data chunk is renamed such that it is a container chunk you will be asked if you want to delete the data in the chunk.

1.11 Binary Editor

The Binary Editor

The binary editor provides an interface for viewing and editing raw data contained within a file. The binary editor window shows a hexadecimal representation of the data on the left, and an ASCII representation on the right. There are two cursors, one in the hex and one in the ASCII representation of the data. The active cursor will highlight the text in white, whereas the inactive cursor will highlight the text in black. The hexadecimal representation is currently limited to displaying the data in longword blocks. Clicking in either of the display regions will move both cursors to the indicated location. If the cursor is dragged with the mouse button held down an area of data will be selected. Dragging the cursor past the end of the window will scroll in the appropriate direction. Pressing

the tab key will toggle the active cursor between the two windows. If the cursor is not visible when a character is typed the window will scroll such that it is. If any data is highlighted when typing is commenced the data will be deleted before the new data is inserted. Typing will either insert or overwrite depending on the insert mode preference. In insert mode typing a hexadecimal character will insert a byte with the left nybble equal to zero. The cursor keys can be used to navigate within the document and pressing shift and cursor down or up will move to either end of the document.

The binary editor window has the following gadgets.

Length: This is a display of the current length of the file in hexadecimal.

Position: This is the current position of the cursor in hexadecimal.

(AP) Entering a new value will move the cursor to the indicated position.

Scroller: The right side of the window contains a scroll bar. Moving the scroller will adjust the data viewed. Pressing the arrows at the bottom of the scroller will scroll the data by one line either up or down.

The binary editor window has the following menus:

Project : See The Project Menu

Edit : The edit menu contains functions for moving chunks to and from the clipboard. See Using the Clipboard for more details on the way the clipboard is handled.

Cut : This will copy the currently selected data to the clipboard and delete it from the file.

Copy : This will copy the currently selected data to the clipboard.

Paste: This will paste the contents of the clipboard at the current cursor position. If any text is already selected it will be deleted.

Navigation: The navigation menu contains options for locating data.

Find : This will open the search requester button.
(AF)

Next : This will highlight the next occurrence of the current
(AN) search item.

Last : This will highlight the last occurrence of the current
(AN) search item.

Preferences: See Preferences

1.12 Using the Clipboard

Using the Clipboard

Most of the clipboard handling in ChunKit is fairly straightforward and automatic, however, because of the low level nature of some of the operations it is sometimes necessary to know exactly how the clipboard handling works. For simple operations, however, no preparation is required.

The Amiga clipboard is designed to use iff format to provide a standard way for applications to send and receive data. Since ChunkIt deals directly with iff files (even executable files are stored internally in iff format) it is a simple matter copy most data to the clipboard. Nevertheless there are complications due to the format of iff files.

An iff file must have either a FORM LIST or CAT as it's outermost node. Thus, if ChunkIt needs to copy one of these chunks to the clipboard it can simply dump it and it's contents. However, if the chunk is a data or PROP chunk it must be placed in a container. The simplest container to choose is the chunk's current parent. Thus if you copy a CRNG chunk that is contained in a FORM ILBM to the clipboard it will copy a FORM ILBM containing the CRNG. When you attempt to paste it back ChunkIt will check to see if you are pasting it into a FORM ILBM. If so it will remove the FORM ILBM container and just paste the CRNG. Otherwise, the whole ILBM will be pasted. (If you pasted it into a FORM ANIM for example). This is usually useful, since CRNG chunks are only valid with ILBM's. This behaviour can be suppressed by enabling the "no container" preference, which will always prevent the container from being pasted. This is useful if you want to copy a data chunk into a container of a different type, without getting the original container embedded in the new.

The process is similar when copying from the binary editor. The data selected will be made into a miniature of the chunk being edited and inserted into a container of it's parents type. This data can then be inserted anywhere into the file and it will have the same parent chunk and data chunk as the original. When pasting to a binary file, the first data chunk in the first form will be opened. This means any data can be copied into the chunk being edited - even if copied from a different program and the data format's are incompatible. Thus you could paste text into an executable to modify the error messages.

There is also a preference in the binary editor to override this behaviour to allow the copying of text. By selecting "copy FTEXT" the data will be copied into a FORM FTEXT with the data being in a CHRS block regardless of the type of the data chunk. Thus, the data can be pasted into a text editor. This option must be used with care however, since the data is not checked for validity and could contain invalid CHRS data. How other applications would respond to having such data pasted is unknown.

1.13 Find

The Find Window

The Find window provides a way to search through data for a given string. When first opened it contains a row of gadgets, a string gadget, and a string display gadget. The data to be searched for is entered in the string gadget and the actual string to be searched for will be displayed in the display gadget. When data is entered into the string gadget it is parsed and the actual search string is displayed in the string display gadget. The data entered will be interpreted in one of several ways:

Context: If context parsing is enabled the string gadget will parse the string in the most likely format for which it is valid. Thus

"123" would be parsed as an int, "\$123" as hex, "1.23" as a float and ""123"" as text.

- Guess: Guess parsing is the same as context parsing except that if the string is ambiguous (i.e.: it could be either a float or an int as in the case of "123") and the last string was one of the possible types it will be parsed with that type. Thus "1.23" will be parsed as a float. Following this "123" would also be parsed as a float.
- Integer: If the string contains only the digits 0 - 9 it will be parsed as an integer. Currently all integers are parsed as long words. The string display gadget will show "Integer = \$<Hex representation of integer>" to indicate that the data was parsed as an int. Signed and unsigned values are allowed.
- Float: If the string represents a non integral number it will be parsed as a floating point in one of several formats. The string display gadget will show either "FFP = \$", "IEEE = \$" or "IEEEED = \$" followed by the hexadecimal representation of the number depending on whether the number was parsed as fast floating point, IEEE or IEEE double.
- Hexadecimal: If the string contains only the digits 0-9 and A-F or is preceded by '\$' or "0x" it will be parsed as a hexadecimal string. Hexadecimal strings may be the full length of the string entry gadget and are byte aligned. This means that if the string "ABC" is searched for, and the data contains the longword "0ABC" it will not be found. The length of the string need not be even, however, so "ABC0" will be found. The string display gadget will show "\$<Hex representation of integer>" to indicate that the data was parsed as a hexadecimal string.
- String: If the string is not a valid member of any of the above formats or is enclosed in double quotes ('"') it will be interpreted as text. The outermost quotes will be eliminated and the string display gadget will show ""<String>"". Note that the string will be enclosed in quotes even if it was entered without them. The outermost quotes will not be a part of the search string. If case sensitivity is active the string will be displayed in all caps.

The search window has the following gadgets when shrunk:

- |« : This will highlight the previous occurrence of the search string and close the search window.
- « : This will highlight the previous occurrence of the search string.
- Expand : This will expand and shrink the search window to display more options.
- » : This will highlight the next occurrence of the search string.
Hitting enter a second time in the string input gadget will have the same effect as this button.
- »| : This will highlight the next occurrence of the search string and close the search window.

The search window has the following additional gadgets when expanded:

- Parse Input: This is a cycle gadget allowing the user to determine how the data will be parsed. It contains all of the different options for parsing the data.
- Convert FP: This determines how floating point data will be converted. If set to FFP the numbers will be parsed in fast floating point format. IEEE will parse the numbers as single precision IEEE and IEEEED as double precision IEEE. None

will prevent floating point conversion from occurring.
Case Sensitive: If checked, all text searches will be case sensitive.

1.14 Bugs glitches and other snafus

1. Bugs

Bugs are errors in the way the program executes and performs. These include errors in the data produced by the program, enforcer hits and most operations the program performs that it was not intended to as designed.

- Copying FTEXT from a binary editor window does not check that the data copied into the CHRS chunk is valid data. This could confuse other programs that expect CHRS to be CHRS.
- Executable blocks are currently not rounded up to longword boundaries. This means that editing a block so that it has an odd number of bytes will allow the creation of a file that cannot be read in again as an executable file. This doesn't apply to NAME or UNIT blocks.
- New format library files will become even more corrupt with this problem. The size of the container LIB chunk won't be written correctly (since the program isn't aware that it is incorrect).

2. Glitches

Glitches are inconsistencies in the normal operation of the program. They're not bugs because they cause no incorrect output to be generated, but they still interfere with the logical behaviour of the program. For example closing all windows whenever the space bar was pressed on a Tuesday would be a glitch.

- The scroller gadget in the BinEd window sometimes leaves strange artifacts. I had fixed this problem by not using GT_GetIMsg in my event loop but when I added the position and offset gadgets I had to reintroduce it. This doesn't have any negative effects but it looks bad. Hopefully this will be cleared up when I implement the binary editor as a BOOPSI object linked to a BOOPSI scroller (if anyone knows about this artifacting please send me a letter).
- INDX blocks for the new format libraries are not parsed, they are simply loaded as one block of binary data.

3. Bug Reports

ChunKit has been thoroughly, but not exhaustively tested, so it is certain that certain bugs remain. I would thus appreciate it if you would mail me a report of any bugs or glitches you find in the program. Since this is the first release there are probably quite a few small errors. Unfortunately I don't have access to another Amiga, so ChunKit has only been tested on an A2000 with an 030 running 2.1. I realize you aren't beta testers, but please try to isolate the problem and make bug reports as descriptive as possible. Saying the program crashed when you click the "new" button tells me little, but saying it crashes every time you click "new" while in a chunk "FORM 666 " tells me where to look for the error. I can't fix a bug if I can't reproduce it (usually). Please include the version number of

ChunKit and if an error message is involved please state it word for word.

In particular I am looking for any of the following errors:

1. Enforcer hits. These are the highest priority since fixing one enforcer hit usually corrects five other bugs. Please include the hunk and offset, the violation and any other information you have.
2. Guru's. Please include the full number and if possible a way to reproduce the error.
3. File corruption. ChunKit should write all files as byte for byte copies of the original. (Consequently it should be able to open it's output in the same format that it was saved.) It may inadvertently fix some errors in the file, but this is not desired behaviour. If you discover a file that ChunKit modifies on it's own please report the type of file and the change (if at all possible send me a copy of the file if it's not private or non-distributable).
4. File format. If a file is modified by ChunKit and then saved, it should be a valid file in the original format. Thus ChunKit should be able to load it again with the same format it originally had. There are currently some exceptions to this rule for executable files. See bugs.
5. Documentation. I would appreciate hearing about any errors or inconsistencies contained in the documentation. The purpose of the documentation is to make ChunKit easier to use and understand, and thus it must be as accurate as possible.

1.15 Things to do

In the next release of ChunKit I hope to include:

- Rewrite the binary editor as a BOOPSI class. This should reduce code size and increase modularity so I can implement new editors more easily. The new class would have a virtual co-ordinate box as it's superclass (a sub class of gadget class) which would be useful elsewhere in the program.
 - Rewrite the memory manager to speed up insertion. For upwards compatibility all access to blocks of memory is through a memory manager. Currently this memory manager is pretty brain dead - it just keeps one big contiguous block and does copy operations for insertion and deletion. A segmented memory manager that stored memory in blocks would speed things up considerably.
 - Write a disassembler for viewing code block's. (This is one reason I'd like to implement the Binary Editor as a BOOPSI gadget).
 - Opening a Symbol should open the relocatable block of the chunk at the designated entry point. This would make it much easier to examine code and data hunks.
 - Write a custom editor for iff struct chunks. By parsing a text file with structure definitions a template could be opened for different iff chunks. This would prevent the internal format of the chunks from being violated and make it much easier to edit many common chunks. I got this idea while reading about a product in comp.sys.amiga.announce. Unfortunately I've never seen the product and forget the author's name. (I would like to take a look at it
-

- since it seems to do most of what I initially wanted ChunKit to do).
- Better type checking for executable hunks. I'd like to make it harder, if not impossible to create illegal hunks and blocks.
- Add an undo function everywhere I can.
- An Arexx port. At first I thought it would be utterly useless to add an arexx port to ChunKit, but I've thought of a few uses for one since. It isn't a high priority and thus won't appear anytime soon.

1.16 Not So Legal Junk

Disclaimer

This bit just says that if ChunKit corrupts your hard disk, your files, or your life I'm not to blame. In other words, you use ChunKit at your own risk. As with any program which modifies the contents of files it is thus advisable to work on a copy to avoid damaging the original.

1.17 Not So Legal Junk

Disclaimer

This bit just says that if ChunKit corrupts your hard disk, your files, or your life I'm not to blame. In other words, you use ChunKit at your own risk. As with any program which modifies the contents of files it is thus advisable to work on a copy to avoid damaging the original.

1.18 Easter Egg

EASTER EGG

I was too lazy to put any Easter egg in the program so here's one in the Docs. If you stripped the ASCII out of the amigaguide you'll find this quite easily. Otherwise you either got lost or had too much time on your hands.

Feel free to skip, disparage, discard or otherwise ignore this. Since only the bored, curious, or otherwise distracted will read this, here is where I can put all of the inappropriate doc things, such as gratuitous ASCII art:

```

      ,____.
      gW@@@8c~+s
      ,W@@W8/~, 'N.
      W@@4@G_t- 'W
      d@WA@Wbg!-, !b
      i@A@@@4Mt(` - Yi
      ]@@@W@WA4\/, ][
      @@W@W@D*Z+Nm_. @
]P8b i@W [-@@-!@KY[ <- Easter Egg
]b8Kmm@WzWmW+==*Ld[
M@@8@K@A[-f -' ,A
]@@@WA8Z,D' .- ][

```

```
'@@@W@@@D]+!      W`  
!@@A@WAN_ ' ' d!  
Y@@@MGG/ ' iP  
V@@@Wb.! gf  
'VM@W_Df`  
~*@@@Af`
```

1.19 About the Author

I am currently a third year student of computer science at the University of Waterloo. I can be reached by email at

progers@undergrad.uwaterloo.ca

for at least the next year, and by snail mail at

Patrick Rogers
2261 Greenwood dr.
Sudbury Ontario, Canada
P3B 1A2

until September, possibly longer.
