ATARI ST

GFA Assembler

GFA ASSEMBLER

Written by: Roland Schutz
Peter Holzwarth

Edited by: George W. Miller
Les Player

Layout by: Andrew Quayle

Translations by: Gunter Minnerup

Published by: GFA Data Media (UK) Ltd.
Box 121
Wokingham
Berkshire
RG11 1FA
(0734) 794941

ISBN 1 85552 003 5

December 1990

Book I GFA ASSEMBLER

Table of Contents

Part III Using The GFA ASSEMBLER

Part IV Additional Information

Appendices

Part I

Introduction

I.0 Introduction to the GFA ASSEMBLER

The purpose of this manual is not to teach Machine Language programming, but to provide a reference manual for using the GFA ASSEMBLER. Many good books are available for instruction in 68000 programming and should be consulted for specific information. However, we do feel that due to its interactive editor, the GFA ASSEMBLER is an ideal tool for both the beginner and the experienced Machine Language programmer. We'll begin by describ-ing the assembler program. We feel this first step is necessary, because many users have already become familiar with various high level programming languages on the Atari ST. The current trend is towards writing either parts of a program, or the entire program in Assembly Language. Many users will find it very interesting to learn a programming language very close to Machine Language.

Machine Language is best defined as the individual bit settings (ones and zeros) which contain the actual information interpreted as instructions by the microprocessor, and executed directly. **Assembly Language** programming is the use of a **symbolic code** in which each statement represents one instruction which corresponds to a Machine Language statement. This symbolic code is easier for humans to read. A single statement in a high level language, such as GFA BASIC 3.0 may include many Machine Language instructions. An Assembler program converts the symbolic code, sometimes referred to as opcodes, into the actual Machine Language instructions used by the microprocessor. In this manual, we will be referring to the process of writing in Assembly Language.

User programs written completely or partly in Assembly Language are characterized by high execution speed and short program length. This is because an Assembly Language program can use the computers memory more effectively. The GFA ASSEMBLER was designed by Assembly Language programmers for Assembly Language programmers. It offers many functions and options that are designed specifically for program development.

Here are a few of the major features of the GFA ASSEMBLER.

The Assembler and Editor are combined in one program. Errors are discovered when they are entered. When you have entered a line into the editor, it is checked for errors and an error message is returned at the line containing the error. No turn around time. If an error occurs during the assembly process you are returned to the editor with the cursor in the line containing the error. With a separate assembler and editor, the source code must be loaded, corrected, saved before you may attempt to assemble it again. Automatic tokenising of the source code. Assembly commands and address types are not saved in plain text, but are converted into short keywords, or tokens. This prevents extra space characters from being saved. This process saves memory in the computer and on the disk and also shortens the loading and saving time required for your source code. (A tokenised source file is only about 50% to 60% of the length of the same file stored as plain text.) Automatic formatting of assembly commands. All input is directly formatted according to the tab positions, which can be set as you prefer. This saves time when entering a program and provides an easier to follow program listing.

Very fast assembly of source code into object files. You will soon discover that the assembler is extremely fast. The editor of the GFA ASSEMBLER can also be used to write programs in high level languages. It offers all the options you would expect in a comfortable and fast text editor. Through it's power to load programs and then to pass this source code directly (that is with no detour through a disk), the editor can also be used quite effectively with compilers.

The powerful debugger, with which programs can be tested, is a stand alone program. Therefore the Editor/Assembler is not excessively large, and the debugger can be used without loading the Editor/Assembler. (Of course the Debugger may be loaded in addition to the Assembler). The debugger is symbolic and screen oriented. The cursor can be positioned on the screen as desired.

Part II

The Editor

The Editor of the GFA ASSEMBLER has its own user interface. This interface is quite similar to that used by GEM programs. However, this interface is faster, while still permitting mouse as well as keyboard input.

The GFA ASSEMBLER can be run in high resolution (monochrome monitor) and medium resolution (color monitor). The GFA ASSEMBLER should not be run from low resolution, as system errors may occur.

In order to understand the operation of the assembler, follow the chain of events that are presented in the following section. If you'd like to streamline your use of the GFA ASSEMBLER, click once on GFA-ASM.PRG from the GEM desktop to select it. Then move the mouse cursor to the Options heading on the Menu bar. Select Install Application. Now enter the default extension .IS so that you may load and run the GFA ASSEMBLER simply by selecting any file with the .IS filename extension.

Please make a backup copy of your original disk and put the original disk in a safe place. Now load GFA-ASM.PRG from your backup disk.

1.0 Selecting a File

Shortly after starting the GFA ASSEMBLER a screen will appear which gives you the chance to select a file to be loaded:

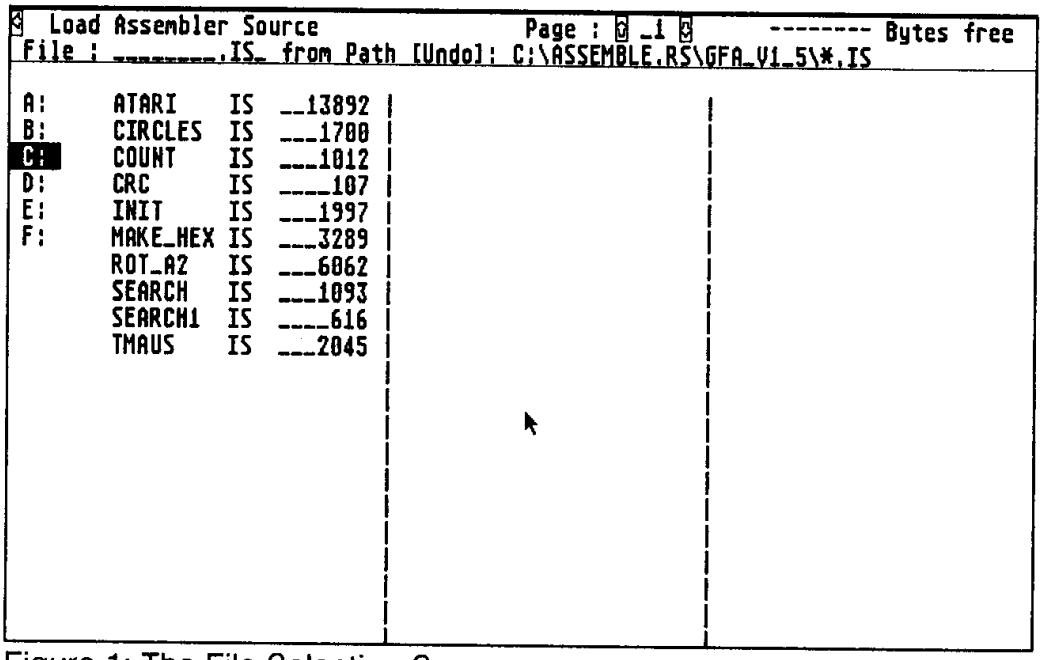


Figure 1: The File Selection Screen.

The Escape Arrow can be found in the upper left corner of the menu bar and has the same effect when the mouse cursor touches this arrow as when the Esc key is pressed. At the moment, nothing will happen because a file must first be opened. If one or more files are in memory, then the file selection screen can be exited by pressing the Esc key. What is happening to the file is described next to the Escape arrow. In in this example the message Assembler Source Load is displayed. If you select a file that was not created with the GFA ASSEMBLER, this message will change to ASCII Source Load as the file is loaded.

The path to the current directory is displayed in the middle of the menu bar. If more then 69 files are in the current directory, an arrow symbol appears next to the page number, at the middle of the upper line of the menu. When this arrow is selected with the mouse cursor, or the Cursor up and Cursor down keys, you can page through longer

directories. When the Shift key is held down it is possible to page through the directory by columns. To the right of the page number is the identification letters of connected disk drives. The current drive is displayed in reverse video.

You can change the current drive by either clicking on one of the other drive letters or holding the Control key and pressing the desired letter key.

To the far right the free memory area on the disk is shown, if there is enough space in the window. If the current drive is a hard disk, the free memory area is only shown after this area has been clicked on with the mouse cursor.

The selected filename is shown on the left side of the second line in the menu bar.Located to the right of the file name is the path to this data. The path can be changed in the following ways:

To the next deeper directory: click on the directory.

To the next higher directory: Press the Undo key or click on the word [Undo] in the middle of the second line with the mouse pointer.

Change the extension mask: either click on the Extension mask (at the right end of the path), enter the new extension and press the Return key, or, when no file name has been selected, simply enter the new extension as the filename extension and press the Return key. In the mask the Wild Card Characters '*' and '?' may be used. If you are in the main directory, the extension mask can be changed to '.*'

The file name can also be changed, by clicking on the desired file in the directory.

Enter the name, access path and extension through the keyboard. You can change the fields for the name and the extension, separated by the period by using the cursor keys to position the keyboard cursor.

When a program is loaded, the wild card extensions '*' and '?' can be used in the file name or extension. You can select one of the file names found on the disk or you can enter a new file name. Press the Return (or Enter) key, or double click on the desired filename.

If you have selected a file name that does not exist in the disk directory, a new file will be created with this file name. The GFA ASSEMBLER will also store the current access path. When you save this file later with the Save function from the File menu or Save Changes from the status screen (see Section 3.1.1), the file will be saved in the proper directory.

The file selector also allows you to select more than one file for loading or deleting. Just press down SHIFT while selecting. Using wildcards, all files conforming to the pattern will be loaded or deleted.

WARNING:

It is possible to delete an entire directory in one go (by entering '*.*').

Now, load the file DEMO.TXT from the root directory of your disk.

2.0 The Editor Screen

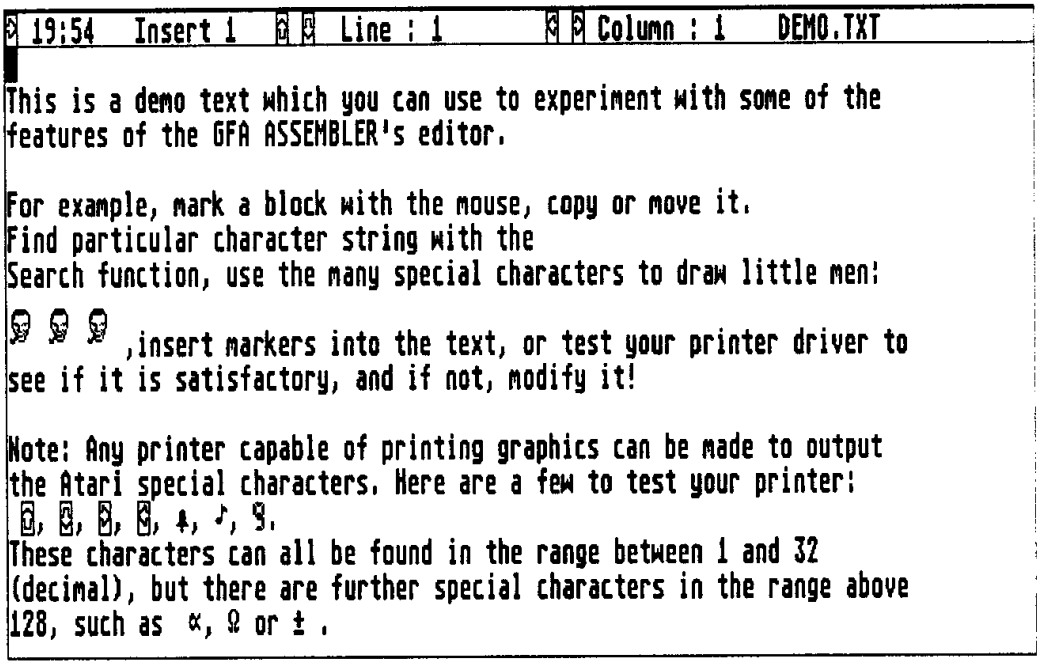


Figure 2: The Editor Screen

If you have previously worked with other editors you'll notice two things when the editor screen appears. First, the cursor blinks. This makes it much easier to locate than with a non blinking cursor. Secondly, there is an extraordinary amount of room available on the screen. Only a single line is used as the status line. The editor always displays 24 lines, each with 80 characters.

In case a line is longer than 80 characters the extra characters will be replaced by an arrow symbol (pointing either left, or right) on the screen. To access the part of a line out of view use the cursor keys (see Section 2.2) or click the arrow with the left mouse button.

2.1 Contents of the Editor Screen

The status line is used to display important information. For example, which line and column the cursor is presently in and which text file is being edited (the number of text files that the editor can contain is only limited by the amount of memory available). In the upper left part of the status line the Escape arrow can again be found, pointing to the right now. This means that you can return to the menu by either touching the Escape arrow or pressing the right mouse button.

You can exit the menu by clicking on the Escape arrow again, by clicking the left mouse button once outside of the status line or by pressing the Esc key. Next to the escape arrow, a digital clock is displayed. This clock displays your system time. It may be set by clicking on the time display or by pressing the F1 function key and entering the correct time. After you have entered the correct time, press the Return key. If you use a clock module, or have previously set the correct time, the correct system time will be displayed.

Beside the time display, the Edit mode is indicated. There are three Edit modes:

Insert 1: Characters, that you enter are displayed at the cursor position.When the Return key is pressed, the rest of the line (following the cursor) will be moved to the next line.

Insert 2: Similar to Insert 1, however, pressing the Return key moves only the cursor to the start of the next line.

Overwrite: Characters entered replace the characters that were previously in that position. When Return is pressed, only the cursor moves to the next line.

You can change between these modes by clicking the mouse button on the mode display, or pressing 'F2'.

By pressing the Shift and Insert keys, it is possible to toggle between Insert 1 mode and the Overwrite mode.

Two arrows follow this display on the status line (Cursor up and Cursor down). Clicking on these arrows with the mouse cursor permits paging through the text, either up or down, one page at a time. You can also page through the text by pressing the cursor up or cursor down key while pressing the Control key. Clicking on the area of the status line which displays the line number or pressing the F5 key permits entering a new line number. You may also enter a label name, up to eight characters in length, or the name of a Macro. The file is searched and if the line is found, or the line which contains the label or the Macro is defined, the cursor moves to the beginning of this line.

When a label/ macro definition has been entered in the status line (after F5) followed by SHIFT+RETURN rather than RETURN, the label/macro is searched for from the current cursor position rather than the beginning of the text.

A letter 'N' appears on the top line, which indicates whether the keypad operates as cursor keys or as numbers. (If the 'N' is present, then it indicates Numbers). See section on keypad keys later in this section for their function.

2.2 Key Assignments in the Editor Screen.

The keys in the keyboard contain the normal key assignments. Additionally, many keys are assigned new functions.

The following keys are used to move the cursor around the screen:

Cursor up:	Moves the cursor up. If the top of the screen is reached, the text is scrolled down one line.
Cursor down:	Moves the cursor down. If the bottom of the screen is reached, the text is scrolled up.
Cursor left:	Moves the cursor to the left. If the left edge of the line is reached, the cursor moves to the end of the previous line.
Cursor right:	Moves the cursor to the right. If the right edge of the line is reached the cursor is moved to the beginning of the next line.
Shift + Cursor up:	Moves the cursor to the upper left corner.
Shift + Cursor down:	Moves the cursor to the lower left corner.
Shift + Cursor left:	Moves the cursor to the start of the line.
Shift + Cursor right:	Moves the cursor to the end of the line.

Home:	Moves cursor to the upper left corner.
Shift + Home:	Moves cursor to the beginning of the file.
Control + Home:	Moves the cursor to the end of the file.

Other new functions:

Control + Cursor up:	Page up
Control + Cursor down:	Page down
Control + Cursor left:	Move one word to the left.
Control + Cursor right:	Move one word to the right.
Delete:	Erase the character directly under the cursor. All characters right of the cursor are moved one character to the left.
Shift + Delete:	Similar to Delete, however if the cursor is at the end of the line, the next line is attached to the current line.
Control + Delete:	Erases the word, including spaces in front of the word, which contains the cursor.
Shift + Control + Cursor right:	Delete rest of line from cursor position.
Shift + Control + Cursor left:	Delete line up to cursor.
Backspace:	Erase the character to the left of the cursor. All characters in the line to the right of the cursor, including the character under the cursor, are moved one space to the left.
Shift + Backspace:	Similar to Backspace, except when the cursor is at the beginning of the line. In this case the current line is attached to the end of the previous line.
Return:	Insert 1 mode: the rest of the line, after the cursor, is moved to the start of the next line. Insert 2 mode and Overwrite mode: the cursor is moved to the beginning of the next line.

Enter: Same as Return.

Insert: An empty line is placed in front of the current line.

Shift + Insert: Toggle between Insert 1 and Overwrite modes.

TAB: Empty spaces fill in from the cursor position to the next tab position.

F1: Set clock (see Section 2.1)

F2: Change between Insert 1, Insert 2, and Overwrite modes (see Section 2.1)

F3: Move screen one text line up. Cursor stays in the present line.

F4: Move screen one text line down. Cursor stays in the present line.

F5: Enter new line number.

F6: Call special character box (see Section 3.2.4).

F7: Repeat search further (see Section 3.4.4)

F8: Erase line.

F9: Call the Macro function box.

F10: Save file assembled in background.

Undo: If you have just changed, or erased a line or a block, a box will appear asking if you would like to recover this information.

Control + Help: The currently displayed screen is sent to a connected printer.

If you have more than one file in memory: (see Section 3.1.1)

Control +.	File change backwards.
Control + ?	File change forwards.

When two text files are displayed simultaneously (see also Section 3.1.1):

Control +]	Enlarge text window.
Control + [	Shrink Screen
Control + '	Right Screen
Control + ;	Left Screen
Escape	Menu bar

Numeric Keypad	While holding the Shift key you can enter the decimal ASCII code of a character and when the Shift key is released the character will be displayed at the cursor position. In this manner you can use all ASCII Characters from 1–255.
Numeric Keys	If you press the Alternate key and simultaneously press one of the numeric keys 1–9 you can place up to 9 marks in your file. These marks can be called by pressing the Control key and then the corresponding Numeric key. (see also Section 3.4.1 – 3.4.2).

The numerial key pad can be changed to function as the cursor keys by using the key combination **Ctrl**+ '**(**' (on the numerical key pad).

The keys have the following meaning:

8: Cursor up

6: Cursor to the right

2: Cursor down

4: Cursor to the left

7: Cursor to the start of the text

1: Cursor to the end of the text

9: Screen display one page up

3: Screen display one page down

0: Insert a line

.: Erase a character

If you are in the Edit Parameter Screen (see Section 3.2.1), then the following key assignments are available. Further functions are available by holding the Control key and pressing the corresponding Character key as follows:

Control + A	Move cursor one word to the left.
Control + F	Move cursor one word to the right.
Control + I	Move to the next tab position (no spaces).
Control + T	Erase the words under and after the cursor position.
Control + C	Page down one page.
Control + R	Page up one page.
Control + Y	Erase one line.
Control + Z	Move cursor to its last position.
Control + S	Save with backup.
Control + U	Load file over present.
Control + B	Set beginning of block.
Control + K	Set end of block.
Control + H	Erase block markings.
Control + V	Move block.
Control + W	Save block.
Control + P	Print the entire file.
Control + D	Print the defined block.
Control + L	Search.
Control + E	Search and replace.
Control + X	Exit the GFA Assembler.

2.3 Mouse Functions in the Editor

The function of the mouse in the Edit screen:

Click with the right mouse button: Menu bar appears. The menu bar disappears when the left mouse button is pressed on the Escape arrow or outside the status line.

For the functions of the mouse outside the status line see Section 2.1.

Outside the status line:

Left mouse button pressed once: Sets keyboard cursor to the mouse pointer position. If the mouse pointer is under the bottom of the text the click is ignored. If the mouse cursor is in a column that has not yet been written in and therefore is longer then the current line length, the cursor will be placed at the end of the line.

Double click on the left mouse button: Jump directly to a line, Label or a Macro. The effect is the same as entering a line number, a Label or a Macro in the status line (see Section 2.1), only here nothing must be entered, except that the line number, the label, or the macro name is found in text.

Hold left mouse button: Marks block. In this way you can quickly mark a block that is contained within view on the screen. The mouse cursor can not be moved outside the current text area while marking a block.

Hold Right mouse button, and press the left mouse button:

Copy block. If a block is marked it is copied to the present position of the mouse cursor.

Hold right mouse button, and double click on the left mouse button: Move block. If a block is marked it is moved to the present mouse arrow.

Depending on the setting of the Edit Parameter Screen (see Section 3.2.1) the editor will react to the mouse arrow touching the upper and lower screen edge:

With the Scrolling function the cursor will roll very quickly in the corresponding direction.

With the function Menu Line the menu bar will appear when the mouse cursor touches the upper screen edge and the corresponding menu is pulled down. The menu is very similar to that of a GEM Program.

2.4 Macro Key definition

Keyboard macro functions can be substituted for often used key combinations. This makes the use of the GFA Assembler easier.

Every event that is executed by the user through a key, the mouse or the menu bar can be changed to a sequence of commands with the macro functions of the GFA Assembler. Every key can be defined eight times (in connection with keys like Shift/Control). With such a definition you can execute with only one mouse click, a key press or a menu call, often used key combinations.

An example for the use of macros would be the saving of all sources before assembling (see examples).

The GFA Assembler saves the keyboard macros internally as a sequence of commands in an ASCII file. Such a sequence of commands looks like the following:

```
PROGRAM event
   Command parameter,parameter,parameter,..
   .
   .
   .
END
```

A list of all possible Events and commands with the parameters can be found in Appendix A2.

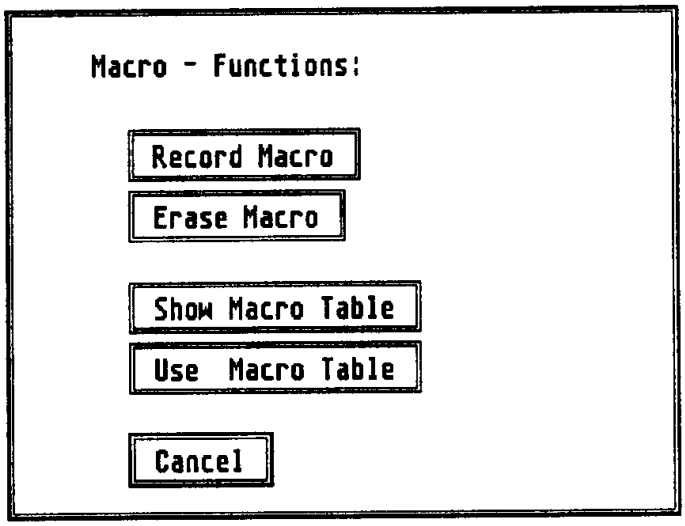
After the starting the GFA Assembler, it searches for the macro table with the name 'GFA-ASM.MON'. If the file is found it will be loaded into memory. If this file is changed during the use of the program, when you quit, it will automatically be saved in its current state.

The macro table may also be edited in the Editor as a text file (see later).

Record Macro The recording of a macro can only be started within the edit screen, whereas it can be ended in any place.

To fill in a new Event you must activate the macro–record–mode and then show the GFA Assembler what it is supposed to do later on.

Press the function key F9 and a box with various macro functions will appear:



In this box click on the function record–macro. Another box will then appear which will ask you to give it an event which is to be changed. To give a key a new function press any key or even a combination of keys (Alt+, Shift+). As mouse events you can exchange the single and double click with the left mouse button. To define a certain menu entry, choose this menu with the mouse or enter it the following way; **Esc,code,code** (for the control of the menu see part 3).

The Macro Functions Box

After you have entered the event, the two boxes will disappear again and in the status line a 'M' will appear in front of the current file name. This 'M' shows you that all events which will be entered from now on will be recorded as the new function for the earlier entered event.

Now enter all the keys, mouse clicks and menu calls. Finally press the key F9 again to end the recording.

Example 1:

The 'Help' key is to start assembling (instead of clicking on the menu item):

press **F9**:	Execute the macro function box selector.
click on **record**	A key, a mouse click or a menu will be the macro.
press **Help**:	the 'Help' key will activate this macro.

Now you program the new function:

select **Menu 6,item 2**:	Assemble function.
press **Return**:	Start assembling with the same name
press **F9**:	End of the macro recording session

Example 2:

Save modified sources before assembling.

press **F9**:	Call the box with the macro function.
click on **record**	A key, a mouse click or a menu will be the macro.
select **Menu 6,item 2**:	Assemble function.

Now you program the new function:

select **Menu 1,1**:	to the status–screen (see part 3.1.1)
Save Changes	click on: all changed sources will be saved.
Return:	back to the text
Menu 6,2:	Assembler–box
Return:	Start Assembly
F9:	End of the recording session.

Hints:

With the macro function it is possible to reprogram every macro event. In the event that important keys or menus are changed it might happen that the assembler will become unusable.

Every key can be exchanged individually, for example, the '1' on the numerial key pad might have a different function to the '1' on the letter key pad.

Erase Macro

You can erase a previously defined macro by clicking in this box. If you entered the previousl example where thje help key starts the assembler, and wish to delete it, when you click in the delete box, you will be asked which event to delete. You would then just press the 'Help' key, and that macro is erased.

Editing Macros

If you want, you can also edit any existing macro. Press 'F9', then click in the macro functions box on the button 'Show Macro Table'. The text 'GFA-ASM.MON' will then appear as the current text in the editor. If no macros have been set up, then the text will be empty. Otherwise you will find a sequence of commands for all the events that exist. You can now edit this text freely. Every change consists with the following sequence of commands.

PROGRAM event	'PROGRAM' starts the change for an event. 'event' is the event for which the change should occur.
Command argument(s)	'Command' stands for a sequence of commands to be executed, when the event nominated after 'PROGRAM' is entered. 'argument(s)' are the possible arguments which may be passed on to the command.
END	Ends the sequence of commands for this event.

Examples:

The prevoius examples might look like the following in a macro table:

Example 1:

```
PROGRAM KEY $00620000
   MENU 6 2
   KEY RETURN
END
```

Example 2:

```
PROGRAM MENU 6 2
   MENU 1 1
   KLICK 600 25
   KEY RETURN
   MENU 6 2
   KEY RETURN
END
```

Further Examples:

```
PROGRAM DOUBLECKICK
   KLICK
   KEY WORDDEL
END
```

A double click will erase a word at the current position of the mouse.

```
PROGRAM KEY F1
   LABEL "1"
END
```

Searches from the position of the cursor onwards the symbol definition '1:' or '.1:'.

```
PROGRAM KEY $033B0000
   TEXT "MAKE_HEX.IS"
   LINE "start"
END
```

When the key combination 'Shift+F1' are pressed the text 'MAKE_HEX.IS' is displayed as long as it is currently in memory. Furthermore if the symbol 'start' is defined in the text, the cursor will jump to that line.

A list of all the commands with their arguments can be found in Appendix A2.

Not every key on the ST-keyboard has a name or a clear ASCII code. For example the key '1' on the numerial key pad and the key '1' on the letter key pad have no name and both have the same ASCII code. To differentiate such keys one can use the hexadecimal codes of a key in the editor macro table. For example the key '1' on the numerial key pad has the hexadecimal value $006D0000 and the '1' on the letter key pad the value $00020000. These values change again if keys like the Shift/Cntrl keys are used.

A new key combination 'Alt+F9' has been introduced so that you will not have to look up the various hexadecimal values of keys anymore (for example in the Tos & Gem book by GFA Systemtechnik). Press any key after 'Alt+F9' – even with a key combination – and instead of the key the hexadecimal value ($qqss00aa) will appear at the position of the cursor. The characters have the following meaning:

qq: qualifier (keys like Alt,Shift, Cntrl)
ss: the scan code of the key
aa: the ASCII code of the key

Example:

The key '1' above the letter key pad has the value '$00020031' after the keys 'Alt+F9' has been pressed. This means that no key like Shift/Alt has been pressed and that the key has the scan code $02 and the ASCII value $31.

Please keep in mind that after the key command KEY either the Scan code with Shift/Alt keys or the ASCII code may be used. The scan code is the only exact way to differentiate between certain keys, for example, the '1' on the numerial key pad. Only this code should be used when you want to differentiate between these two keys within a keyboard macro.

Examples:

```
1)      PROGRAM KEY "1"
           KEY "No longer present!"
        END

2)      PROGRAM KEY $31
           KEY "No longer present!"
        END

3)      PROGRAM KEY $00020000
           KEY "No longer present!"
        END

4)      PROGRAM KEY $006D0000
           KEY "No longer present !"
        END
```

In the examples 1 and 2, both the '1' on the numerial key pad and the '1' on the letter key pad have been changed and instead of the character '1' , 'No longer present !' will appear. In example 3 only the '1' above the letter key pad and in example 4 only the '1' in the numerial key pad are changed.

To make a changed table accessable for the Assembler, click in the macro functions box, on the button 'Use Macro Table'. The text will then disappear from the editor window and the text that was there before will reappear.

With the possibility of editing macro tables you are in the position to modify your changes afterwards, erase them or even create new changes.

Furthermore you can choose between various macro tables by loading them as if they were ASCII files and then pass them on to the editor, using the button 'Use Macro table' in the macro functions box.

3.0 The Menu Bar

The menu of the GFA ASSEMBLER is similar to the menu of GEM Programs. It has the advantage of only being displayed when it is needed and it is managed by the mouse as well as the keyboard.

You can call the menu from the Edit screen by either pressing the <Esc> key, passing over the 'Escape arrow' with the mouse, pressing the right mouse button once, or if you are in the Edit Parameter Screen (see Section 3.2.1) making the function 'Menu Line' active by moving the mouse arrow against the upper screen edge.

The arrow in the menu bar points to the left, this means that you can leave the menu list by clicking on this arrow or pressing the <Esc> key.

You can pull down the individual menus by touching them with the mouse cursor, pressing the corresponding function key (found before the name of the menu item), or pressing the appropriate number key. The file menu will appear, if you point the mouse arrow to the word 'File', press the function key <F1> or the numeric key '1'. It doesn't matter if you use the number from the numeric keypad or from the keyboard itself. During the time a menu is shown the 'Escape arrow' points upward, this means that this menu can be exited when the <Esc> key is pressed or the 'Escape arrow' is touched by the mouse arrow.

You can go directly to another menu in the same fashion as with any GEM Program, move the mouse arrow to the desired menu item. You can select the individual menu entries either with the mouse or with the corresponding function key or numeric key. There are ten menu entries in the 'File' and 'Block' menus. In order to reach the tenth menu entry, either press the <F10> key or the '0' on the numeric keypad.

The 'Atari' menu header, represented by the 'Fuji' symbol, occupies a special position within the menu system. It is only possible to activate it by using the mouse cursor. In this menu you can activate the menu item 'GEM Fileselect' (with a check mark in front) or deactivate it (no check mark). If this item is active, a GEM Fileselect box will appear

instead of the normal file selection screen of the GFA ASSEMBLER. When you exit this box the box will disappear immediately.

3.1 The 'File' Menu

The File menu contains items for selecting functions which manage the loading and exchange of data between GFA ASSEMBLER and the disk drives, RAM disks and any hard disks connected to the system.

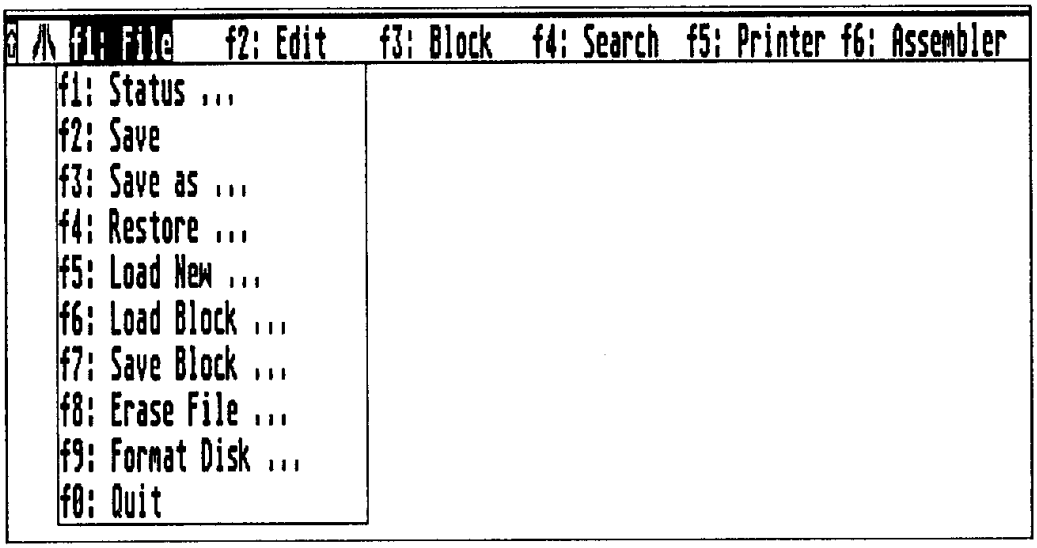


Figure 3: The File Menu

3.1.1 Status

The Status screen informs you about the file that is in memory. You can also change to other text files or select to display two text files.

In order to try out the functions of the Status screen, load an additional text file into memory. For example, load ASSEMBLE.BS.

Call the Status screen (by selecting 'Status', from under the 'File' menu, or by pressing <F1>) The following screen will now appear:

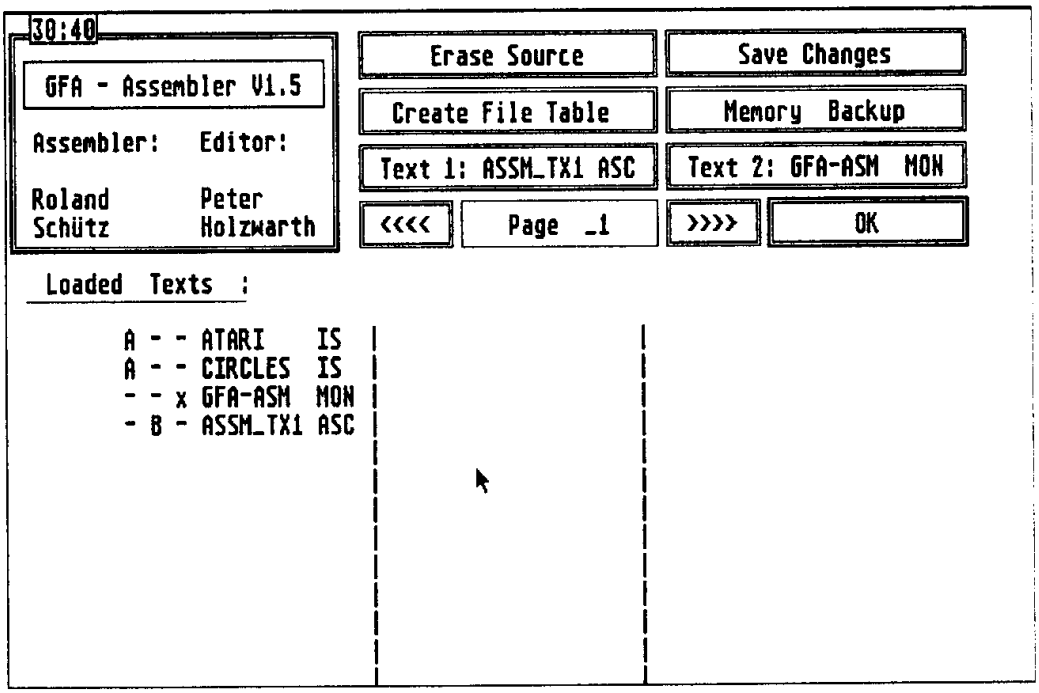


Figure 4: The Status Screen

The present time is displayed in the upper left corner of the Status screen. To the right are the 'buttons'. The buttons can be selected by using the mouse.

In the lower part of the screen are the names of the files in memory. Letters are placed in front of the names. Each letter has a specific meaning as follows:

A: The file is assembly source. It is compressed in memory.

B: This file contains the 'Global Block' (see Section 3.3.9).

x: Changes have been made to this file which have not been saved to disk.

If you now want another file to be displayed, simply double click on the name of the desired file. The status screen is then exited and the file is displayed.

If you want to know how much memory area is occupied by a file, click once on the file name with the right mouse button. An Alert box will appear which can be exited by clicking on the 'OK' button or by pressing the <Return> or <Enter> keys. The file in memory requires approximately 15 to 20 per cent more than the same file stored as a file on a disk.

In order to determine how much memory is still free, click on the Copyright field 'GFA ASSEMBLER 1.x' (x represents the present version number). An Alert box similar to that used for the file memory area will appear.

The buttons used for the status screen are:

Erase Source

Select a file name (click on a file name once with the left mouse button) and then click on the button 'Erase Source'. The selected text segment will now be erased from memory. (However, previously saved versions will exist on the disk.) There is no verification question before the source is erased. You should only click on 'Erase Source' when you are sure you really want to erase the source.

Save Changes

All text, that has been changed, and not yet saved to disk in it's new form, is saved. You are given the chance to change disks when you

have loaded files from various disks using the same disk drive. The GFA ASSEMBLER records the disk identification number and name of each file that is loaded. You should always give each disk you will be using to store GFA ASSEMBLER files a name and record this name directly on the disk. Files that were loaded from a RAM disk, or a Hard disk will also be written back to that drive. New files will be saved in the directory that was last shown (see Section 1).

Create File Table

A file selection screen will appear in which you can select the drive and path for saving the assignment table. The assignment table contains all text that is presently in memory. Files that are loaded from disk will also contain the identification number and name of the disk.

In order to use the assignment table, you must install the filename extension '.TAB' as an application for the GFA ASSEMBLER (see Section 2). Then you can click on the file table from the GEM Desktop and GFA ASSEMBLER will load and then immediately load the appropriate file. You will be prompted to change disks as required.

Memory Backup

All loaded text, that is in ram will be backed up, you can select where to back up to via the fileselect screen.

Text 1,Text 2

If you would like to display two text files side by side enter the names of both files in the fields (click on the desired filename, and then on the text box). To return to the Edit screen, click on the 'Ok' button or press the <Return> key.

The file specified in Textbox 1, is displayed on the left side of the screen, the file specified in Textbox 2, is displayed on the right side of the screen (in order to display only one file at a time, double click on the desired filename).

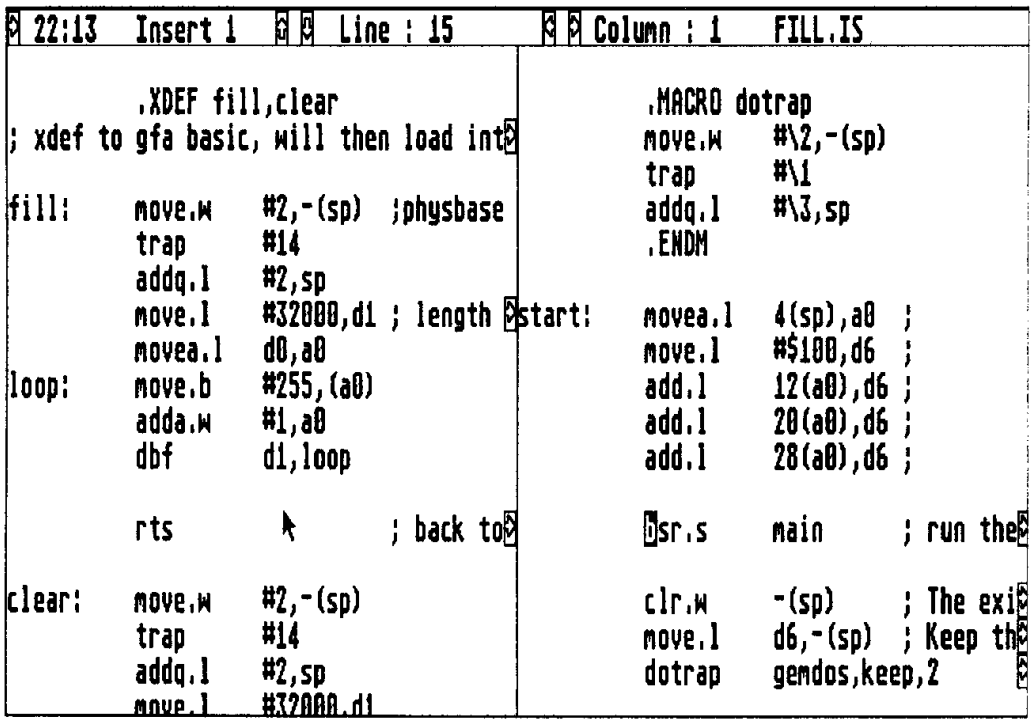


Figure 5: Two text files in the Edit Screen

Visualize the cursor in Figure 5 as blinking in the left text area, and stationary in the right. This means that you can work with the text in the left side of the screen (because the left text is active), the text on the right side is only being presented. The status line will always contain the name of the text that is active at the moment in the upper right side of the screen.

Page x

When there are more then 56 text files in memory you can page through the contents directory by clicking on the '<<' and '>>' buttons. The line 'Page x' will always display the page that is presently being pointed to by the directory. The number of text files that can be in the memory at any one time is only limited by the amount of memory present. You can write large programs in small sections and not need to save the entire program when a small change is made. If you are writing a mixed program, that is, a program written partly in Assembly Language and partly in a higher programming language, all parts of the program may be in memory at once.

Ok

You are returned to the Edit screen by either clicking on the 'Ok' button or pressing the <Return> key.

3.1.2 Saving Text

The current file is saved in the directory from which it was loaded. The disk from which the file was loaded must be in the drive. There is no prompt for the proper disk to be put in the drive.

3.1.3 Saving with a new name

A file selection screen will appear on which you can determine where the current file is to be saved to. Enter a new name for the file. The path may also be modified, if so desired.

3.1.4 Restore File

When you have made changes to the current file and feel that you have made too many errors, you may choose to start over by loading the last saved version of the text file and replacing the present file.

CAUTION: If the memory is so full that the correct file will no longer fit in the available memory area, the present file will be erased, but no replacement will be made.

3.1.5 Loading a New File

Loads an additional text file or creates a new text file (see Section 1). After the file has been loaded, the new file will be displayed on the Edit screen.

3.1.6 Load Block

Loads a block of data at the present cursor position. This data will then become part of the present file. The loaded file will be marked as a block (see Section 3.3). It does not matter if the present file or the file to be loaded is in ASCII Format (in clear text) or in Assembler format (compressed).

3.1.7 Save Block

This function saves a marked block to a disk. A file selection screen permits the name to be used for saving the file to be entered.

3.1.8 Erase Data

Data may be erased from a disk with this function. This may be necessary if you do not have enough space available on disk to save the file you are working on. The file selection screen is displayed until you return to the Edit screen.

3.1.9 Format Disk

This function permits the formatting of a new disk. Remember to specify a disk name. This name will be used by the GFA ASSEMBLER (see Section 3.1.1).

3.1.10 Exiting the GFA ASSEMBLER

This function exits the work area and returns to the GEM Desktop. If any work has not been saved, an Alert Box will appear giving you the opportunity to save the file before quitting.

3.2 The 'Edit' Menu

Functions that have a direct effect on the editing of text are gathered together under the Edit menu.

3.2.1 Edit Parameter

When you call this function, the following screen will appear:

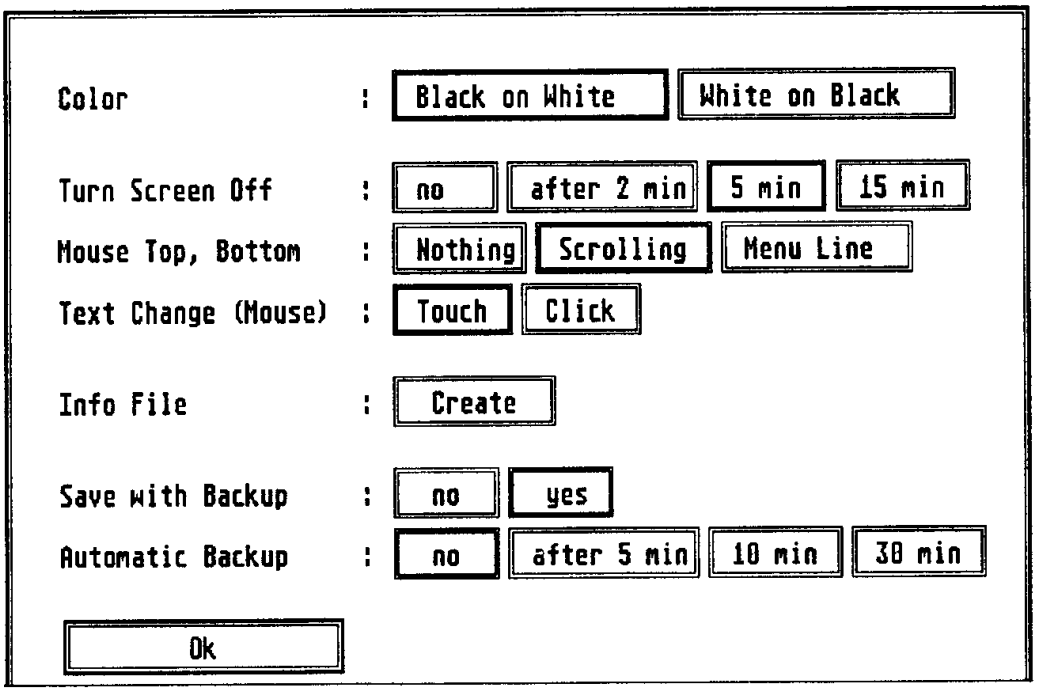


Figure 6: The Edit Parameter Screen

If you have a colour monitor you will see a colour palette displaying the colours selected at this time in the upper part of the screen. Monochrome users will see buttons labelled 'Black on white' and 'White on Black'.

You may select the display colours by adjusting the RGB values for the individual palette registers, or by clicking on the appropriate buttons (for a monochrome system). Change the RGB value by clicking the arrow symbol located to the right of the displayed palette buttons.

You can also move the mouse to the corresponding R, G, or B value

by pressing the cursor keys (cursor left, cursor right by pressing the cursor up and cursor down keys.

The next line permits choosing automatic screen save, and the time duration after which the screen should shut down. This function protects the monitor from having an image 'burnt in' by a lack of input over a long period of time. If 'after 2 min' is set and no input has occurred for two minutes, the screen will turn itself off. As soon as any key is pressed (or the mouse is moved) the screen will turn on.

CAUTION:If you have a monochrome monitor and less than 32 KBytes of memory remain, this function will not operate.

The next line determines what happens when the mouse cursor is moved to the upper or lower edge of the screen display. When the 'Scrolling' is set, the cursor will move quickly in the corresponding direction. If 'Menu Line' is set, the menu line will appear, including the corresponding menu entry. This makes the management of the program very similar to that of a GEM Program.

The key combination <Control> + 'Letter' can be used to switch the keyboard between the standard key assignments and command table key assignment (see Section 2.2 and Section 3.2.7).

Clicking on the button labelled 'Info File : Create' causes the GFA ASSEMBLER to automatically create a data file called 'GFA-ASM.INF' in which settings of the buttons, the contents of the input folders and the tab settings are saved. This data is automatically loaded each time the program is again, making it unnecessary to set these parameters each time.

If the 'Yes' button is selected following the 'Save with Backup' entry, the previous version of the file will be given the extension '.OLD' when a file is saved. This protects the old file from being overwritten. Finally, if there has been no new input after changes have been made to a text file, the new file can be automatically saved to disk after a specified period of time. Only files that can be saved without requiring a disk change will be saved.

3.2.2. Text Attributes

In the text attribute menu, important adjustments can be made for each single text. The most important one is probably the choice if an assembler source or just any other source is supposed to be edited.

Assembler Source/ASCII Source

The editor of the GFA Assembler can be used in ASCII mode for general sources for any ASCII text. It offers a familiar and fast text editor.

In the assembler source mode the editor offers further functions which are especially set for the development of assembler sources:

Intelligent Input of Assembler Programs

During program development, bugs can occur not only because of errors in logic, but also during the input and correction of individual processor commands. The most common error sources are:

Using prohibited address types. For example: move.w d0,4(pc)

Over writing word boundaries. For example: moveq #300,d0

Using prohibited extensions. For example: adda.b symbol,a1

Forgetting an operand. dbra d0

Syntax. Making a typing error when entering a command, operand, or calculation

Inconsistency during line entry. As soon as a line is ended by pressing the <Return> key, the file is checked for errors and saved in a tokenised format. If an error is discovered, the incorrect command will be displayed with an error message. The user is not required to correct the error immediately. (The error could be caused by an incorrect label which must be checked in another source code before a correction can be made.)

If a program is assembled with these types of errors, they will be displayed again by the assembler. If a line is entered properly, it is

automatically set to the proper tab positions. Each line in Tokenised format has the following principal format (however, all named components must not be present in each line).

By using the character 'i' (ASCII 173; accessible on the keyboard with <Control>+ <0>) many such structures can be placed in a single line as long as the maximum line length of 255 characters is not exceeded.

Attention must generally be paid to ensure that:

- Assembler Mnemonics are recognized in upper and lower case and any combination of cases (For example Move Operand). It is not possible to give a macro the name of an assembler command.
- Symbols and Macro names can automatically be set to upper or lower case depending on your preference. This option can be set in menu 6,1 (see part C, Section 1.1).
- Label definitions (max. 16 characters long) must always end with a colon (':', ASCII $3A). A double colon ('::') is a global definition of the symbol. The double colon marks the difference between label definitions and macro calls. Some assemblers do not insist on this distinction. Included with the GFA Assembler is a utility to convert their source code appropriately.
- Mnemonics (Processor commands) must be written out in full in order to distinguish from similar Macro calls. For example, if you want to automatically convert 'mo' to 'move', then a macro with the name 'mo' can never be used. The following conversions are automatically executed during input:

add, sub, and, or, eor	->	addi, subi, andi, ori, eori
add, sub	->	adda, suba
move	->	movea

If you'd like to reduce the amount of typing required further, you may assign Mnemonics or entire Assembler Commands to the keyboard according to your individual preference (see Section 3.2.1).

The most frequently used Mnemonics extensions for the specification of the work‡width can be omitted. The Editor will automatically insert the command specific Default extension. These will be the most commonly used extensions For example, 'move' will be given the extension '.w', changing it to 'move.w'. – The identification of value overflow and incorrect addresses plays a role with limiting the Macro programming possibilities (see the example below).

Example:

```
(a) move.w  symbol,#4       ; can be reported with the input
    move.w  symbol,\parm    ; the address passed by '\parm'
                            ; can first be checked when the
                            ; program is assembled
                            ; (compare to writing a macro.)
(b) add.w   #100,d0         ; is corrected to
    addi.w  #100,

or:
                            ; during input
    add.w   \parm,d0        ; can be corrected during assembly.

(c) addq.w  #9,d0           ; is reported as illegal during input.
    addq.w  #9 symbol,d0    ; can first be checked when the
                            ; value of 'symbol' is known.

(d) move.w  (a0) (a1),d0    ; immediately recognizable
    move.w  \parm(a1),d0    ; first checkable during assembly.

(e) move.b  #1000-1,d0      ; is first reported during assembly
                            ; because the editor
                            ; carries out no calculations.
```

In general, you could say that the editor identifies the errors that the perfect user would also be able to recognize during program input. The deletion of a single line could have far reaching results during assembly, especially with complex program segments. This is more than likely the exception rather than the rule, but the help offered by the automatic correction feature should not be underestimated.

One problem area is a typing error that can not be seen as incorrect by the computer.

Example:

Incorrectly entered symbols or Macros.

Incorrectly written Mnemonics with no extension which could be mistaken for a Macro call.

Label definitions without ':' are treated as Macro calls.

Error filled brackets in expressions are silently executed during a calculation.

A good check is that the displayed line will show the line in the proper format after tokenising.

The following are declared as register names (in upper and lower case):

d0 d1 d2 d3 d4 d5 d6 d7

a0 a1 a2 a3 a4 a5 a6 a7

r0 r1 r2 r3 r4 r5 r6 r7 (Equivalent of d0-d7)

r8 r9 r10 r11 r12 r13 r14 r15 (Equivalent of a0-a7)

sr ccr usp

The complete Motorola 68000 command set is implemented.

Character combinations that are unrecognisable to the editor are interpreted as comments and are marked with a semicolon and placed at the end of the line.

We suggest you experiment with the practice file to in order to become familiar with this input system.

Note:GFA ASSEMBLER Source code should always be saved to the disk with the extension '.IS' to make identification easier.

Assembler sources can be converted without any problems from one mode to the other by pressing the correct button and OK/RETURN in the following box. If you want you convert a text for example from the assembler mode to the ASCII format please click on 'ASCII Source' and confirm your choice. After a short while the text will be turned into ASCII text.

The other way round you can convert ASCII Text (for example an assembler source which you have earlier on produced with a different editor) to source for the GFA Assembler by loading the ASCII text into memory (the GFA Assembler recognizes automatically that it is an ASCII text) and then clicking on the button 'Assembler Source'.

Spaces – Tabs in the text

The GFA Assembler editor only supports only text that use spaces to separate words. Some editors also use tabs (ASCII 9). To be able to continue using such files and to be able to produce text with tabs, a function 'Source uses Spaces' and 'Source uses Tabs' exists.

In case there are tabs instead of spaces in a text file, you can obtain spaces by choosing 'Source uses Spaces' and all the tabs are substituted by spaces. The current tab adjustment (Menu 2,3) is used to make the spaces.

Please set this before the reformatting of text according to your wishes. The other way round all spaces in the text are replaced by tabs if you decide to access 'Source uses Tabs', the current tab adjustment setting is used.

MS DOS – AMIGA DOS

The ASCII format of Commodore Amiga files differs from that of MS DOS for the separation of lines. The MS DOS sequence is 'cr+lf' whereas the Amiga only uses 'lf'. The ASCII character set of the Amiga also differs a great deal from the MS DOS standard in the use of the special characters (for example German 'Umlaut'). The GFA Assembler is able to convert the character sets between each other. When converting characters which do not appear in the destination

character set, an upside down question mark appears in the source code ' '. The character conversion to MS DOS format is executed as soon as the correct choice has been selected. The character conversion to Amiga Dos format is only done when saving the source code. It might happen that when saving longer texts that there is a short delay before the drive goes on.

3.2.3 Tabs

The tabs can be defined freely over a width of 254 characters in preset steps. In the bottom part of the screen a set area of 4 lines appears. The first line stands for the first 80 columns, the second for the next 80 columns etc. The vertical lines serve for orientation. You can set tabs using the mouse by using the buttons to set or clear tab icons, or by clicking directly on the column. If you want to set tabs using the keyboard you can use the following keys:

1:	Erase all tabs
2 to 9,0:	Sets tab icons, the key '0' sets the tabs in steps of 10.
Shift+numerial key pad:	Set/erase single positions. The input is done as with the special characters: Keep the shift key pressed and enter the column.

After the confirmation of the new choice of the tabs, the assembler source will be displayed in the newly formatted way.

3.2.4 Special Characters

In the text, which you produce with the GFA Assembler, you can use every character of the Atari character set (from ASCII 1 to ASCII 255). You can obtain the characters not present on the keyboard either through the numerial key pad (see part 2.2) or using the special characters box.

This box can be called either by using menu 2,4 or by pressing the F6 key. It will always appear in the part of the screen where the cursor

is not. This way you will always be able to watch the line in which you want to insert the special character. You can choose a special character through the mouse by moving the mouse onto the special character and then clicking the left mouse button. Through the keyboard you can move the arrow of the mouse with the cursor keys 'Cursor up', 'Down', 'Left' and 'Right' through the special character box. With the 'Insert' key the special character will be inserted under the mouse.

The special character box appears after every other key input.

3.2.5 Text Comparison

When you display two texts at the same time on the screen (see part 3.1.1) you can let GFA Assembler compare them automatically. The comparison will take place from the position of the cursor onwards. It does not make a difference whether or not one of the texts or even both texts are in the assembler mode. When a difference is found the cursor will jump in both texts to the corresponding position.

3.3 The 'Block' Menu

The Block Function enables moving, copying, erasing, and passing information between text blocks.

Every text file in memory may have an individual block marked. The starting and end points of the block can be set as desired. The defined block area is displayed as underlined text.

3.3.1 Setting Block Start

The starting point of a defined block is set at the present keyboard cursor position. The block stretches to the point where the block end is marked. If no end block point has been defined, the block runs to the end of the text. To remind you a starting point for the block has been defined, a check mark will be placed in front of the word 'Set Start' and the block will be shown as underlined.

3.3.2 Setting Block End

The end point of a block is set at the present keyboard cursor position. If no block start point is marked, the block will begin at the start of the text. To remind you an ending point for the block has been defined, a check mark will be placed in front of the word 'Set End' and the block will be shown as underlined.

Note: Defining a small block which is shown within the screen display can be down more quickly with the mouse. (See Section 2.3.)

3.3.3 Copy Block

A marked block is copied to the present cursor position. If the cursor is within the block, the instruction is ignored.

3.3.4 Move Block

A marked block is moved to the present cursor position. If the cursor is within the defined block, the instruction is ignored.

3.3.5 Erase Block

A marked block is erased. This operation can be 'undone' with the 'UNDO' key.

3.3.6 To Block Beginning

The keyboard cursor is moved to the beginning of a defined block.

3.3.7 To Block End

The keyboard cursor is moved to the end of a defined block.

3.3.8 Remove Mark

The block markings are removed.

3.3.9 Declare Global Block ←

When a block is declared as a 'Global Block', it may be copied to other text files in memory. A check mark is placed in front of the menu item 'Block > global', and the character 'B' is displayed in front of the current text on the Status screen (see Section 3.1.1) to remind you of a global block declaration.

3.3.10 Use Global Block

A block which has been declared 'Global' can be copied into the currently active text file, beginning at the cursor position.

3.4 The 'Search' Menu

The functions found under the 'Search' menu are used to quickly find specific text position.

3.4.1 Set Mark

Setting and jumping to a mark allows moving easily to a frequently used section of text. Marks can be set in each text file in memory, independent of other files in memory. In order to set a mark, a box with a list of the 9 available marks will appear.

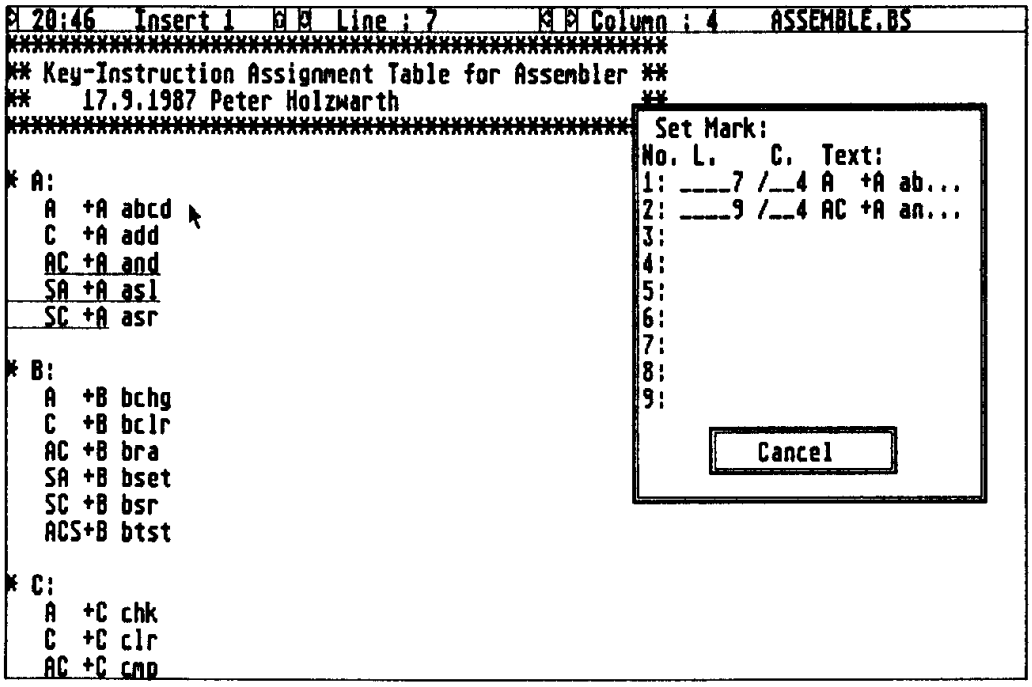


Figure 8: The box 'Set Mark' with two set marks.

Marks that are already set are identified with the line, column and the first eight characters after this position. In order to set a new mark at the current cursor position, either click the left mouse button at the corresponding line in the box or enter the number of the mark with the keyboard. You can use a Numeric keypad number or the normal keyboard numbers.

3.4.2 Go to Mark

You can move directly to a marked text segment from the box 'Go to Mark.

Note: Marks can also be set and jumped to directly from the keyboard. See Section 2.2 for instructions.

3.4.3 Search Character String

A specified character string may be searched for. If the string is found, the cursor will be positioned at the start of the found string. This is somewhat comparable to jumping directly to a label definition (see Section 2.2).

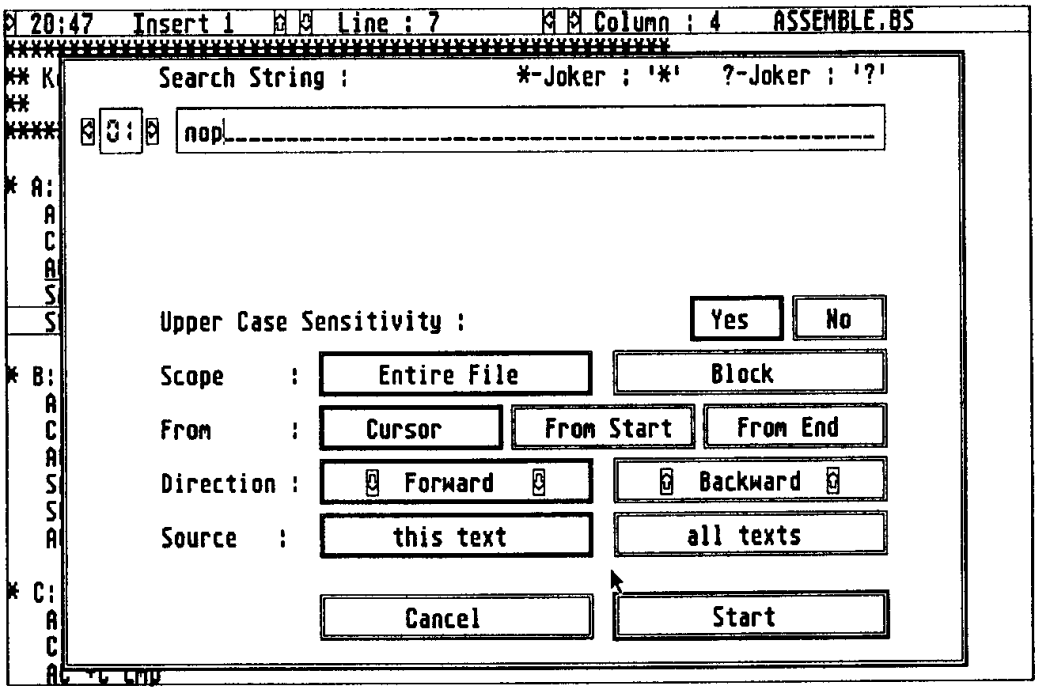


Figure 9: The Box for 'String Search'.

The character string is input in the box as shown in Figure 9. There are 20 such input areas used in order to limit the number of times an frequently used character string must be entered. These input areas may be switched to by clicking with the mouse cursor on the arrow symbol located to the left of the input area. The number of the current input area is shown in the box between the arrows. The search string

can consist of any character from ASCII 1 to ASCII 255.

The characters '*' and '?' (or depending on preference additional characters) are reserved for use as wild cards. The character at the position in the string of the '?' wild card is ignored. If you use the string 'h?llo', the second character is not compared and character strings such as 'Hello', 'Hxllo', 'H7llo' and so forth will be found.

When the '*' wild card is used, any number of characters will be ignored. For instance, the search string 'The meal * good' finds strings such as 'The meal was good' and 'The meal was not very good' will be returned.

To use the characters '*' and '?' as characters in a search string, define other characters as wild card characters for the search routines. This is done by clicking the left mouse button on the expression '*' and '?' and entering the new character. Both wild cards may be used in the string as often as is desired. The meaning of the buttons to differentiate between Upper and Lower case letters is obvious.

Also no buttons sequence which is logically incorrect can be activated. For example, searching backwards can not be initiated from the start of the text area.

For text, the options 'This text' and 'All text''are available. If the 'All text' button is active, the current text will be searched from the cursor position for the search string first. If not found, all other texts in RAM (beginning with the one immediately following the current text in the status screen directory) will searched. The status line always displays the name of the text being searched.

If the search string is found within the given search area the cursor will be placed at the appropriate position. If the expression is not found, the cursor remains at its present position.

When a search operation is in progress, a check mark is placed in the menu in front of the 'String Search' entry.

3.4.4 Continue Search

'Continue Search' is a way to continue searching for a specified search string beginning at the cursor position. This function is only accessible when a search or a search and replace was previously executed (a check mark must appear in front of the appropriate function).

3.4.5 Search & Replace

This function permits a search string to be replaced by another expression. An input box is used to enter the text:

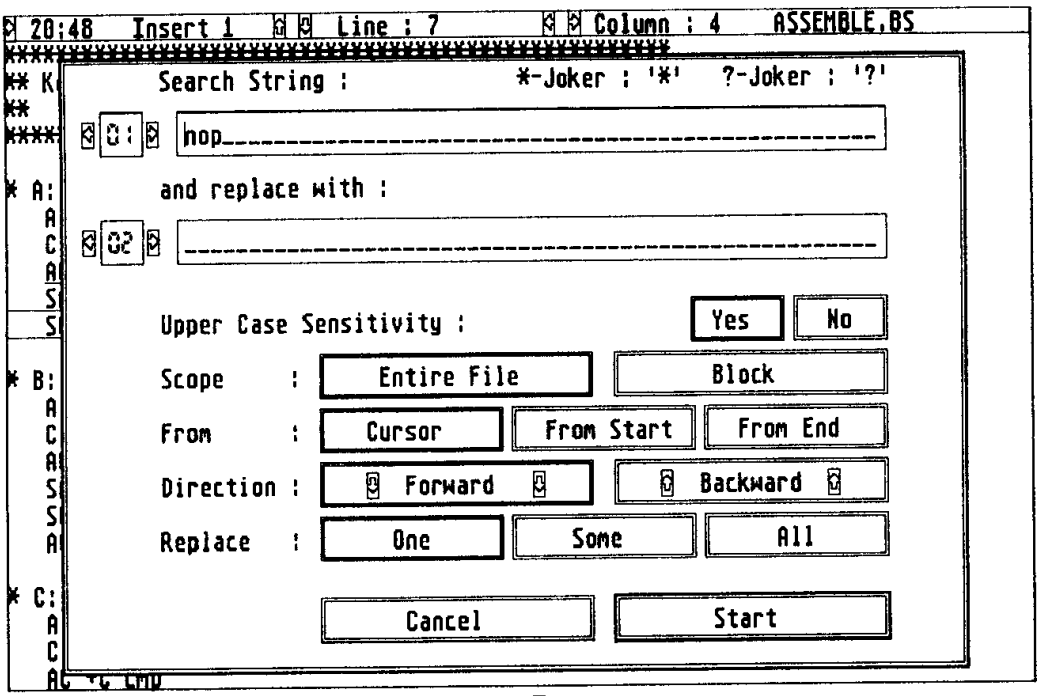


Figure 10: The Search and Replace Box.

There is an additional input field in this box in which the expression that is to replace the first expression can be entered. Wild cards are not permitted in this expression. This means that the replacement expression must always be as it is in the input field. There are also three buttons in this box that determine how often the replacement should occur. When you click 'Some', an inquiry will be made each time the search string is found and before the replacement is made.

3.4.6 Jump to Text Beginning

This function (or with the key combination <Shift> + <Clr/Home>) moves the cursor to the beginning of the text segment.

3.4.7 Jump to Text End

This function (or with the key combination <Control> + <Clr/Home>) moves the cursor to the end of the text segment.

3.4.8 Jump to the Previous Cursor Position

This function returns the cursor to the position in the text where it was located before a jump through the text. (For example, after a Label search, Character string search, or jump to a mark.).

3.5 The Printer Menu

Within this menu entry are the functions for controlling the parameters for the control of the printer. While characters are written in the Spooler–Buffer (see later) or when the spooler is turned off, the printing procedure may be interrupted by pressing the 'Esc' key. As soon as all the characters are in the Spooler–Buffer the printing procedure can be interrupted (Pause) or cancelled (Stop) by using the printer parameter box.

3.5.1 Printer–Parameters

A box will appear for the adjustment of the page format of the spooler and of the character–converter and configuration table:

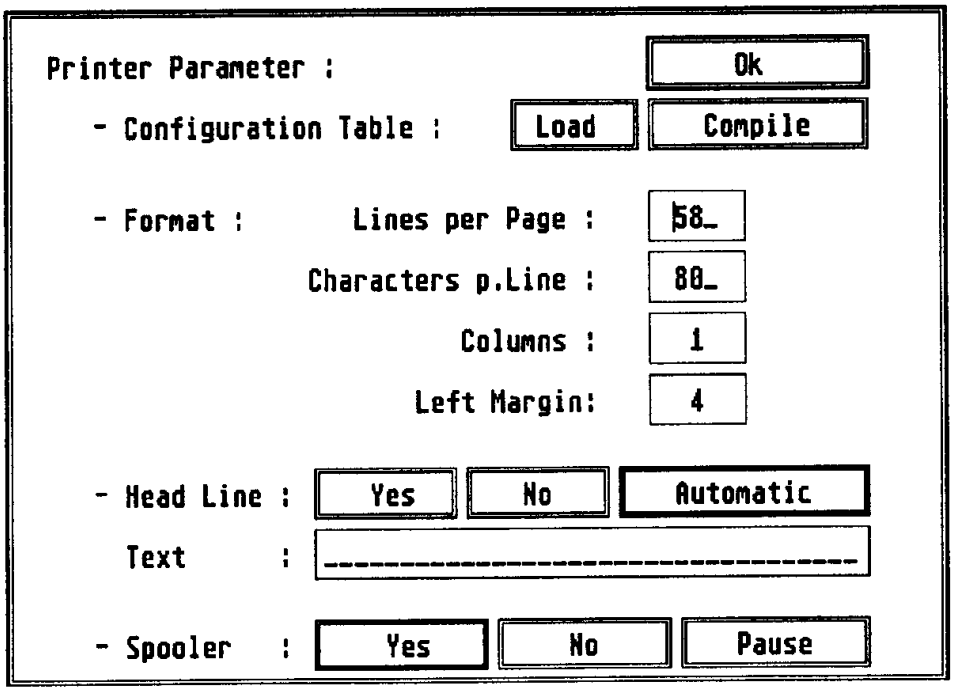


Fig 11. The Printer Parameter Box

The configuration table which you could have provided in the first line contains the information which makes it possible for the GFA Assembler to communicate with various other printers.

The configuration table is usually prepared in an ASCII format in ASCII source mode and compiled using the button 'Compile – Configuration table' to a far smaller '.CFG' file and then stored. For the format of the ASCII file see appendix A4.

When booting the GFA Assembler the table 'PRINTER.CFG' is searched for and then automatically loaded into memory. You should give the table of your standard printer this name. If no such file is found, the GFA Assembler will print with its standard adjustments (see appendix A3).

When you are working with various printers you can load and replace the standard table by other '.CFG' tables with the function 'Load – Configuration table'

In the input area for the page format you can adjust the paper size, the size of the border on the left where it's not to be written on and the amount of columns which are supposed to be printed next to each other. The printing of 2 or 3 columns lying next to each other can become very handy especially with very long assembler listings. This will result in shorter printings and a better outline. To print more columns adjust your printer to its maximum number of characters that can be printed per line. A value that I would suggest is for example, for assembler listings with 136 characters per line, 2 columns will fit nicely next to each other.

To give your output its own headlines, activate 'Headline' by clicking on the 'Yes' button. Following this you should enter your headline, which is to be at the top of your page in the input field. GFA Assembler can also produce headlines automatically if you wish. Simply click on the button 'Automatic'. In this mode the name of the file, the date, the time and the page number will be printed at the top of each page of the currently printed text, not minding if it is the entire text or just a block.

To be able to send characters directly to the printer (for example adjustments for the type of print, the amount of characters per line etc.) simply enter in the input for the headline the corresponding order of the characters and then do a double click on the input field.

In the last line of this box you an activate or deactivate the printer-spooler which is integrated in the program. This spooler is

able to print while you continue to edit your texts. The size of the spooler is dynamically looked after by the GFA Assembler, this means that the spooler-buffer is always the size it needs to be. This spooler is set out so that it will not create any problems with laser printers or with the use of a hard disk as it does not run on an interrupt, it is more or less something like multi-tasking. This means that it prints while the editor is waiting for an input. You can't print while assembling or other such time consuming tasks.

You can interrupt the printing procedure temporarily through the button 'Pause'. The entire printing procedure is stopped using the 'No' button and then the spooler reserved memory is set free again.

3.5.2 Printing the Entire Text

The entire text is printed according to the settings in the printer parameter box.

3.5.3 Print Block

The block area is printed according to the settings in the printer parameter box.

3.5.4 Print with line numbers

The desired area is printed with line numbers.

3.5.5 Print without line numbers

The desired area is printed without line numbers.

3.5.6 Print all pages

All pages ot the dersired area are printed.

3.5.7 Print Odd Pages

Only the uneven pages of the desired area are printed.

3.5.8 Print Even Pages

Only the even pages of the desired area are printed.

With these last two options, it is possible to print firstly the odd pages, turn the paper over, and then print the even pages. This enables printing a listing on two sides of the paper.

Part III

Using The GFA ASSEMBLER

1.1 Parameter Settings

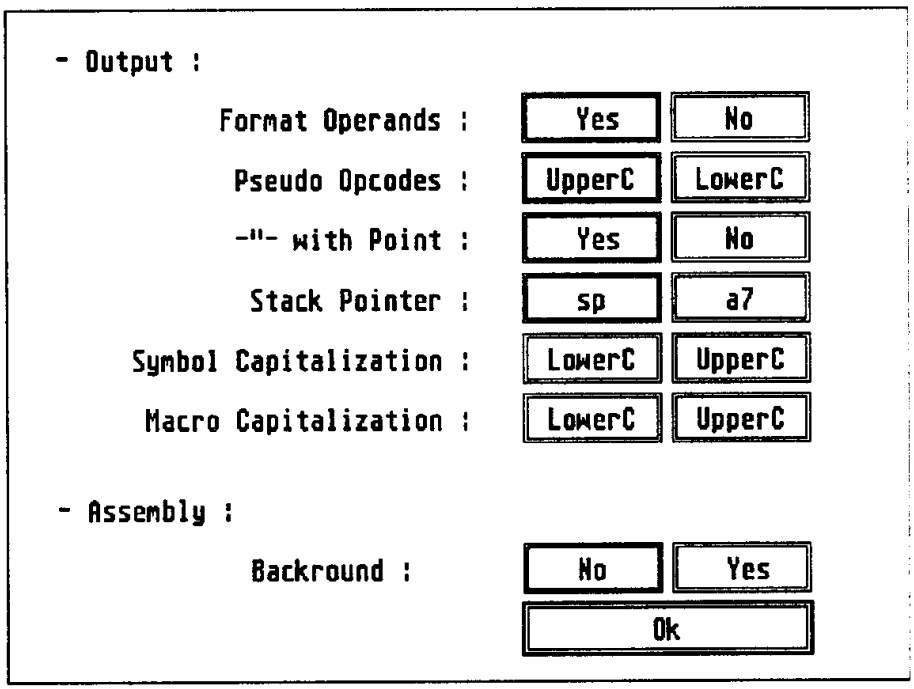


Figure 12: The Assembler Parameters

The settings accessed through the Assembler Parameter menu, shown in Figure 12, affect the output of the tokenized Assembler Source code. It is also possible to select whether the name of the text file to be saved should be the same as the source code loaded or if it is to be selected by using the fileselect menu.

Format Operands

Output/suppression of a tab between an assembler mnemonic or Macro call and it's operand.

Psuedo Opcodes (UpperC or LowerC)

Pseudo opcodes, are used by the assembler and have nothing in common with the mnemonics or the machine language command set of the processor; output choice between upper and lower case.

Pseudo Opcodes (with point yes/no)

You can direct the output to put a full stop in the pseudo opcode field.

Stack Pointer (sp/a7)

The Stack pointer (sp/A7) determines if the abbreviation 'sp' (stack pointer) or the register identification a7 is to be written.

Symbol Capitalisation (Lower/Upper)

Symbol text style determines in upper or lower case should be used for symbol names. According to individual preferences, symbols can be in changed directly into upper/lower case.

Button:	Lower	Upper	Function
	out	**out**	**no conversion**
	out	**in**	**to upper case**
	in	**out**	**to lowercase**
	in	**in**	**no conversion**

Macro Capitalisation (Lower/Upper)

Macro text style (lower/upper) is the reverse of 'Symbol Capitalisation'.

Background Assembly (No/Yes)

If the 'Yes' is active the source code will automatically be assembled during any inactive user periods. The user will notice very little slow down when the assembly is in progress.

If there are no errors with the assembled text, it can be saved by pressing <F10>. The following information will appear in front of the current filename during this process:

>> multitasking assembly in progress

V ready to be saved

? Break because of an error (n)

Note: The advantage of this pseudo multitasking assembly is that unused time periods during input of the source code are used to the fullest advantage. If the text encountered by the automatic assembler contains instructions not yet in the source text, they are assembled when this text is found in memory. Otherwise the assembly will be stopped. (The user would have to pause for lengthy load times while working with the editor otherwise.)

If the user changes the text of the current Source code during the on line assembly, the assembly is immediately stopped and during the next user pause it starts over from the beginning.

Use Source Name

The name assigned to the assembler created Source code is assigned according to the extension rules. The user can set the extension according to need as explained in the next section.

After the assembly is completed, the fileselect menu is displayed to specify the desired filename. Because the file path of the assembler is not known to the assembler during it's work (Programs, Object codes, Protocol, Cross-reference) the files will use one of the following temporary file names:

GFA_TEMP.PRG	**for programs**
GFA_TEMP.O	**for object modules**
GFA_TEMP.LST	**for Assembler protocols**
GFA_TEMP.REF	**for cross-reference lists**

The files are placed in the directory of the drive that the Source code file was loaded from. Eventually, files of the same name will be destroyed, as newer versions of the files overwrite older versions.

When the final file name is determined, the data is renamed and placed in the desired directory. The drive or hard drive may not be changed.

1.2 Assemble

The assembler is called using the Assembler menu item 6,2. It can only be started if a tokenized text file is in memory in the active window. The operation is as follows:

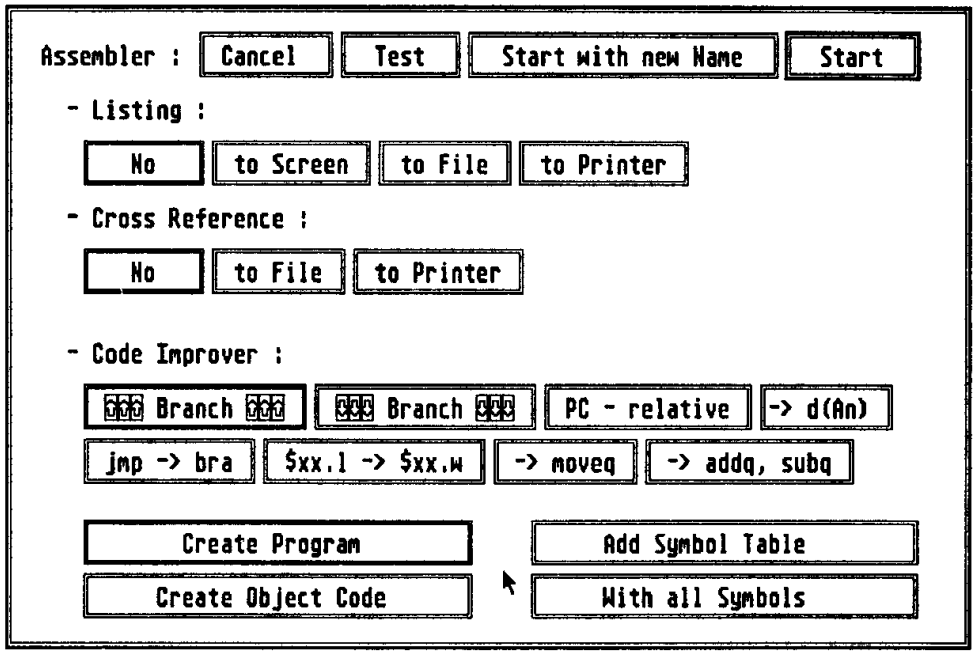


Figure 13 The Assembler Menu

Explanation of the individual buttons:

Listing

This function sets the output medium for the assembler. The default setting is 'No' which menus that all listing protocol directives in the text are ignored (see Section III 2.1 Pseudo Opcodes).

Cross-reference

Cross-reference lists are helpful during debugging to provide and maintain an overview of large programs. The reference list uses all defined symbols, Macro calls, and .Include files of the Source code and records the line number which accessed the call. This shows which routines are called most frequently and permits optimization of

those areas.

Files with the extension '.REF' and the printer are available as output devices for cross-reference lists. The page format depends on the parameters set by the Printer Parameter Screen. The line width is always 80 characters per line. There are two operating modes reserved for the creation of a cross-reference are:

a) During the assembly

b) Separate call from the Assembler menu entry, '**Cross-reference**'

For Cross-reference Lists created during an assembly, the value of the symbol and its attribute word are named. The symbols and macros from the .Include files are also included. An entry in the cross-reference list has the following format:

Symbol definition

Symbol name line number/include number {value and attribute} {line with references}

Variable/abs. value

Symbol name line number/include number {value and attribute} {line with references} (If the value of the variable is changed with 'set', the state at the end of the assembly is given.)

Macro definition <Macro name> Line number/ .Include number

{line with call}

(The line numbers following a slant line are the numbers of the appropriate source text: 0 means Main text, every other higher number gives the .Include file in which the line can be found).

Optimising

1) Branching is handled in two ways:

The Short version is one word long and contains the line to be jumped to in the Low Byte with a relative 8 Bit offset. This means only jumps of, from -128 to +126 bytes from the current program counter are possible.

With the Long version, the Offset follows the command opcode as a separate Word. The jump distance permitted in this case is from -32768 to +32766 Bytes. This variation is more powerful but requires more memory and execution time because of the required bus access.

It is best to use the Short version when possible. The computer will automatically decide if this is possible. For this the address of the line to be jumped to must be known. It is simple to determine if the short jump is within reach if the address is lower than the address containing the branch instruction (backward optimization).

2) If the address is higher than the address containing the branch instruction, then the assembler must assume that the line to be jumped to is not within reach. If the address is later found to be within reach of by a short jump, it can be optimized. This will shorten the jump instruction command by 2 Bytes and the entire text following the optimized command must be moved forward by 2 Bytes (forward optimazation).

It is possible that because of one optimization the entire program could be come more compressed.

3) Absolute Jump (jsr, jmp) can be changed into a PC Relative 16 Bit Offset Variation (bsr, bra).

4) Another form of optimization is changing an absolute length address into a PC relative address. This requires:

a) The address must be within reach (16 Bit Offset, compare this with conditional branching).

b) The destination(PC) address must be permitted with the command to be optimized.

c) If object code is created for the linker, the addresses in question must be found in the same text segment.

With the PC optimization, jsr and jmp commands to an absolute address concurs with the jmp->bra optimization. The PC optimization has preference.

Addresses that are found in the first and last 32 KBytes of the address area are absolute short addressed.

5) The address is not a long word as with the absolute long addressing, but rather a word. Before the bus access it is expanded to 32 Bits by the processor. With this type of addressing the ATARI ST has a much faster access time. If desired all non relocatable long word addresses can be converted from absolute short by the assembler.

6) Addition and subtraction commands (addi, adda, subi, and suba) are converted to addq and subq commands as long as the first sum is not smaller then 1 and not larger then 8. Correspondingly, the command:

move.l #xx, Dn is converted to

moveq.l #xx,Dn when: 128 <= xx <= 127.

All of these optimizations are implemented in the GFA ASSEMBLER. The default setting, which we will refer to as the 'development mode' does not create very compact program code, but this setting is very useful during the development of a program. When you have completed testing and developing your program, select the desired level of optimization.

7) Absolute addressing –> indirect address register

When addressing constants and variables in assembler programs, you often have to use absolute–long addressing. These have two drawbacks, however: their length of 4 bytes per address and their slow speed.

To avoid these disadvantages, data areas are frequently integrated into the program segment and PC–relative methods of addressing used. Unfortunately PC–relative addresses are only permitted with source operands.

There is an alternative in using the '**indirect address register with offset**' method. This is permitted wherever an absolute–long address would also be possible. The base of the data section to be addressed is loaded into an address register and the individual data are accessed via word offsets. If a program is completely converted to this method, up to 40% processing time and storage space can be saved. Data sections which are far apart can then, if necessary, be addressed via different address registers. Should you not be able to spare enough registers for this, however, you could carry out this optimisation in parts of the program only.

The conversion to indirect address register addressing, however, can distort a source text to the point of unreadability. Further more, calculating the offset to the address in the base address register is not always easy.

The GFA Assembler offers a method which enables the user to carry out this optimisation more comfortably:

> Sections of the source code can be surrounded with the directives
>
> .BASE register number, base_address ...; source text...–.
> and .ENDB register number. All absolute addresses of memory cells which can be found in the same segment (cf. the SECTION instruction!) as the 'base_address' and are not more than +–32 kbytes removed from the latter,

will be converted to indirect address register addresses with offsets. The offset is calculated automatically; the address register used will be the one specified in 'register_number'. The user will have to ensure for himself that the 'base_address' value is actually found in the address register specified at the time of program execution. Several optimisations can be performed simultaneously with regard to different address registers (also different SECTIONS!). (Nesting of .BASE/.ENDB structures!).

To activate the optimisation, the user has to additionally click on the appropriate button (d(An)) in the assembler call menu 6,2. Thus the user has the option to generate non-optimised program versions with unchanged source code (for symbolic debugging, for example). If .BASE and .ENDB instructions are found in the source and the assembly is without optimisation, the assembler will warn the user with appropriate messages.

The new optimisation can only be used in conjunction with the branch-forward-optimisation.

Example program:

```
Code optimisation with          | Manual  optimisation
.BASE instructions:             | (Same result)
Activated optimisations:        |
Branch forward, PC-relative,    |
d(AN))                          |
                                |
.TEXT                           |.TEXT
lea BASIS_1a5                   |lea BASIS_1,a5
movea.l #BASIS_2,a6             |move.l #BASIS_2,a6
.BASE 5,BASIS_1                 | ;
.BASE 6,BASIS_2                 | ;
move.w Symbol_1,Symbol_2        |move.w Symbol_1-BASIS_1(a5)
                                |Symbol_2-BASIS_2(a6)
tst.1 Symbol_3                  |tst.1 Symbol_3-BASIS_1(a5)
beq Label                       |beq.s Label
bclr#2,Symbol_4                 |bclr#2,Symbol_4-BASIS_2(a6)
Label:                          |Label:
jsr Symbol_5                    |jsr Symbol_5(pc)
nop                             |nop
nop                             |nop
Symbol_5:                       |Symbol_5:
rts                             |rts
.ENDB 5                         | ;
.ENDB 6                         | ;
```

```
        .SECTION 6,BSS
BASIS_1:
Symbol_1: .DS.w 1
Symbol_3: .DS.l 1
        .SECTION 7,DATA
BASIS_2:
Symbol_2: .DS.w 1
Symbol_4: .DS.b 1
        .END
```

Note:

The assembler makes two passes while in the development mode. The short form branch can be specified with the '.s' extension with forward branches. The programmer must ensure that the branch instruction is within limits (otherwise an error message will appear). The extension is ignored during forward optimization.

Only the optimization of forward branches requires intensive amounts of time (Because of the required address changes).

All additional optimization requires no additional time. Only in combination with forward branches will the time required be increased. (Because forward references are examined.)

Branches in other sections are not normally optimized. If a reference contains an import, it will not be optimized.

PC optimization can be forced. During programming, data areas are replaced by text segments. (Access of data is handled through absolute long addresses.)

Program / Object Code

If no unverified references appear in a source code (for example through modular programming with include files) directly executable programs can be created.

In other cases object code is generated in DRI format. More information about object code is included in the section of this documentation which discusses the linker.

If you are using a symbolic debugger, it is useful to use a dependent symbol table. Instead of incomprehensible hexidecimal addresses, the symbol will appear during disassembly. More information is available in the section of this manual which discusses the debugger.

A symbol table is always contained in the object code for the linker. We can also determine, except for Export and Import, that the extra symbols are placed in these tables so that they available later in the

executable program. The choice between only global symbols and all symbols is also allowed in the symbol table for the executable program. Information about the action of the assembler during the assembly process will be displayed in a window. Access to data files is shown with the path used.

The upper of the two windows ('MESSAGES:') contains only information for the user. The information in the lower window ('Warnings:') is for errors encountered during the assembly process.

There are two categories of errors during the assembly process:

Fatal Errors

Fatal errors lead to an immediate halt in the assembly. An attempt to include a source file that is not available is an example of such an error.

Warning

A warning does not hinder the assembly of the source code and is shown only to inform the user of the progression through the assembly process. When a warning is encountered, the output medium (screen, printer, data file) can be determined by a key press. Error messages, as well as the possible cause of the specific error, can be found in the appendix.

If an assembly is ended prematurely, the assembler waits for a key stroke so that all of the messages on the screen can be read. When the <Return> key is pressed, the user is placed in the Edit Mode.

1.3 Cross-reference

A cross-reference list can also be created separately. There must have been no assembly accomplished. As was described in the section about assembler calls, the values and attributes of the symbol and the include data can not be supported in this operation.

The output can be to either a printer or a data file (with the same path as the source file). The data file will use the extension '.REF)'.

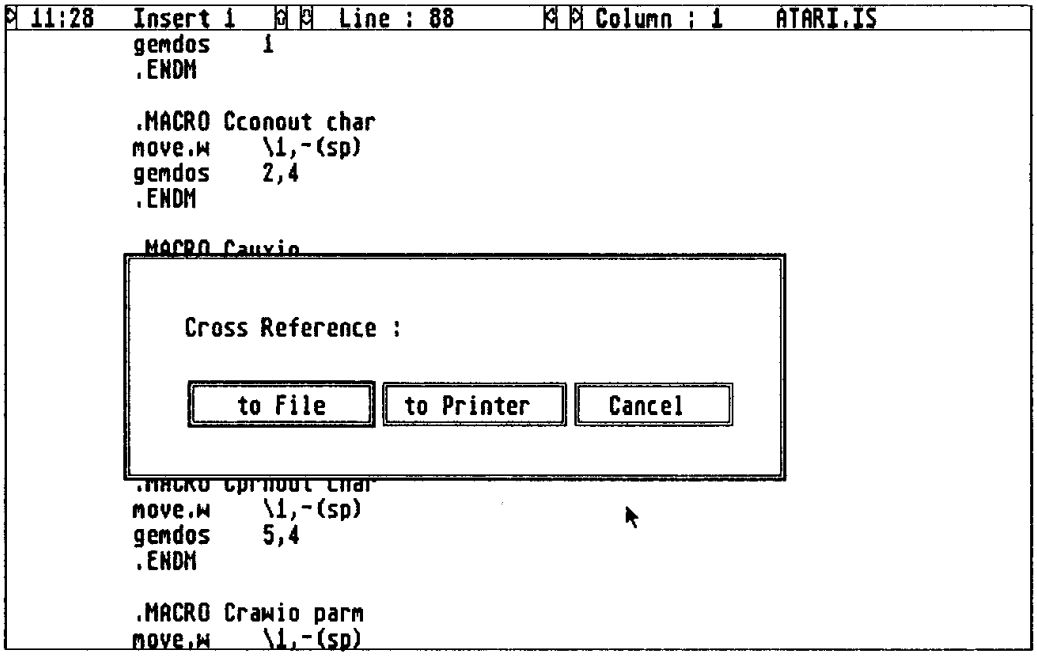


Figure 14: Cross-reference Box with three output device selections.

2.0 Source Code Directives

2.1 Pseudo Opcodes

2.1.1 Program Structure Commands

An assembly program is organized into three logical segments:

- Text Segment (contains program code)
- Initialized Data Segment
- Non-initialized Data Segment

The user determines in which of these data types program parts will be assigned with the corresponding directive when writing the Source file. It is also possible to further divide each of these three segments. Programs in coded form can be arranged differently than that of the source file. This makes structured programming easier and increases the degree user friendliness. It is also possible to simulate field structures from higher languages through a fourth virtual segment which can define the offset for the access path of a data file.

The GFA ASSEMBLER has the following commands for the defining program structure:

.TEXT

The following program code belongs to the text segment.

.DATA

Inserts an initialized data segment.

.BSS

Beginning of a non-initialized data area. The .BSS area of a program is not saved with the program. It is marked in the program header, however. If a program is loaded, the operating system will reserve a corresponding memory area at the end of the .DATA segment.

NOTE: .DS instructions can be found at the end of .BSS segments.

.ABS {Offset}

The .ABS segment is a help point for the programmer and is not relevant to the assembler operation. It is similar to the .BSS segment. It simply serves to define absolute symbols and requires no memory area.

The optional parameter, 'Offset' determines an offset for all additional definitions in the .ABS segment. The default setting for the offset is zero.

The structure can be defined as in the following example by using the .DS Directive. The program counter in these segments increases itself automatically by the size of the reserved area from symbol definition to symbol definition.

If the .ABS segment is omitted, the running value is not lost. When the next .ABS segment is opened, the defined offset is used, as long as none other has been explicitly given.

Example 1.1

```
                .BSS
control:        .DS. w 14       ;Work area for VDI
ptsin:          .DS.w 12
                .ABS 0
Opcode:         .DS.w 1         ;Offset in control Array
Point_in_ptsin: .DS.w 1
Point_in_ptsout:.DS.w 1
Length_intin:   .DS.w 1
Length_intout:  .DS.w 1
Functions_Id:   .DS.w 1
Device_Id:      .DS.w 1

X_coordinate:   .DS.w 1         :Offset in ptsin Array
Y_coordinate:   .DS.w 1
                .DS.w 2
Radius:         .DS.w 1
                .DS.w 1

                .TEXT

                ;circle middle point (200,100) and radius 75
                ;under VDI drawing


Draw_circle:    lea.l           control,a0
                move.w          #3,Point_in_ptsin(a0)
                clr.w           Point_in_ptsout(a0)
                move.w          #10,Functions_Id(a0)
                move.w          dehandle,Device_Id(a0)
                move.w          #200,X_coordinate(a0)
                move.w          #100,Y_coordinate(a0)
                clr.w           X_coordinate+2(a0)
                clr.w           Y_coordinate+4(a0)
                move.w          #75,Radius(a0)
                clr.w           Radius+2(a0)

                jmp             VDI
```

(In that example, the passing field is initialized with fast indirect addressing. This increases the readability of the assembler source code.)

.CARGS {#offset_1} ,Symbol_1.w(, symbol_2.1}{ ,#Offset_2}{ ,Symbol_3.w)

The .CARGS command is a more compact form of .ABS Segment. It influences the program counter of an eventual .ABS segment. If it is used within an .ABS Segment, this segment is isolated, the .CARGS command is executed and the .ABS segment is continued with the program counter of the .CARGS command. As Symbol Extensions only '.w' for a word length and '.l' for long word may be used. The program counter will be increased by 2 or 4 Bytes respectively.

Only even expressions may be used as offsets. If the first expression following .CARGS' is not an offset, then 4 will be used as the default. This is because of the possibility of using the .CARGS command for passing parameter offsets on the stack, as it is used in the C programming language.

Example 2.1

Function to determine string length. The string address is passed through the stack. The return jump address of the routine must be passed, also.

```
String_length:  .CARGS    .string
                move.l    .string(sp),a0;.string=4
                moveq     #-1,d0        ;default length
.l:             addq.l    #1,d0         ;reads to
                tst.b (a0)+             ;'0'=end
                bne       .l
                rts
```

Example 3.1

The structure definition from Example 1.1 can also be accomplished with the CARGS command:

```
            .BSS
Control:    .DS.w 14
            .CARGS#0,opcode.w,point_in_ptsin.w,point_in_ptsout.w
            .CARGS#6,Length_intin.w,length_intout.w,Functions_ID.w
            .CARGS  #12,DeviceID.w
pt_sin:     .DS.w 12
            .CARGS#14,X_Coordinate.w,Y_Coordinate.w,#8,Radius.w

            .TEXT
            etc.
```

Example 3.2

To get parameters passed to the program via the stack:

```
.CARGS      #13*4+4,source.l,dest.l,width.w,height.w
.TEXT
movea.l     source(sp),a0
movea.l     dest(sp),a1
movea.w     height(sp),d0
movea.w     width(sp),d1
```

would result in the following code:

```
movea.l     $38(sp),a0
movea.l     $3c(sp),a1
movea.w     $42(sp),d0
movea.w     $40(sp),d1
```

.SECTION Number,Segment_Type

The .SECTION command is the easiest variation of the segment definition. It enables setting up of 64 program segments with independent program lines (Number 0 to 63). Each section will be assigned to one of the four segment types (.TEXT, .DATA, .BSS, .ABS, etc.). The Assembler sorts all used segment types according to type and number during the assembly process. That is, the first text sections will be assembled (in ascending number order), then all data sections, and finally all BSS sections. The user is not required to continually give the segment number, the segment number can be specified as a macro passing parameter.

Each segment may only be assigned to one type. If a segment is exited and then later returned to, the assembler will continue with the old program counter value. (Exceptions are: .BS segments.)

Branches (Bcc) and PC relative addressing between the segments are allowed, they are not optimized because of structural reasons.

Please note:

For reasons of compatibility with later versions on different computers, the SECTION instruction permits a further string parameter following the segment_type. It will be ignored during execution.

Hints:

Because the operating system only differentiates between three segments, sections about the assembly operation have no effect. They are, however, an element of structured assembly programming.

The sections 0 to 3 are predefined with the commands .TEXT, .DATA, .BSS , .ABS and can only be accessed with the corresponding type.

.TEXT corresponds to	.SECTION 0,TEXT
.DATA	.SECTION 1,DATA
.BSS	.SECTION 3,BSS
.ABS (offset)	.SECTION 4,ABS {offset}

.END

The .END command placed at the end of the source code is optional. In a program file, it causes all following instructions to be ignored. .END also marks the end of .INCLUDE files.

.LOCAL

Local symbols are valid only between two non-local symbols. Local may be used to separate an area of local symbols without defining a non-local label (More about symbols will be explained later.)

.ORG section_number, base_address

Until Version 1.2 inclusive, the generation of absolute code already adapted to a fixed memory address was only possible via the detour of the Linker-ABS instruction. This option is now available at assembler level with the ORG instruction. The ORG instruction is SECTION-orientated, so that a distinct base address can be specified for each program segment used (up to 64).

Only finished programs, not object modules, can be generated. Imports and exports are not permitted. Undefined symbols are assigned the value of the corresponding SECTION base. NO PC-RELATIVE REFERENCES may be made between absolute program segments. A base address must be specified for each SECTION (or TEXT,DATA,BSS) used. All SECTIONs without such a base address will be ignored. The program code generated has no header. Instead, a header file containing the program name and the .HD extender are written.

This header has the following structure:

```
SectionBaseAddress.l    ; ie. 2 longwords for each SECTION with
SectionLength.l         ; ORG instruction
...
SymbolTableLength.l     ; Length of symbol table, if used.
```

.GLOBL, .EXTERN, .PUBLIC, .XDEF, .XREF

These symbols are synonyms for the global declaration of symbols. If the assembler is not creating an executable program, but is generating object code for the linker, all symbols are imported or exported with one of these command declarations. (The global definition of labels is also possible through a double colon ('::'), compare this to the chapter about symbols.). XDEF exports a label, and XREF imports a label.

.COMM symbol,expression

This command applies only to the linker. If object code is created, the .COMM specified symbol is used as an Import and contains the symbol status 'common'. The linker checks all object modules for such 'common' symbol status. Each of these symbols is assigned a value according to expression during it's assembly. During linking the 'common' symbol is converted to a .BSS segment and supplied an appropriate address. The following non-initialized data area is as many bytes long as the largest of the values passed with expression supplies.

With this option .BSS areas can be used within many object modules.

2.1.2 Macros and .Include

Macros are the most important tool for structured assembly programming. They allow many assembler **Mnemonics and Pseudo opcodes** to be assembled under one function name. A macro can be called from anywhere within the program with parameters, as with other higher level programming languages. If the programer has many macros that have been previously created, assembly language programs may be written very quickly by using an **.Include** file to load them during the assembly process. The **.Include** directive is the 'big brother' of the macro call and makes possible the binding together of entire source codes during assembly. By using the .Include file for macro and variable definitions. the user can create his programs in

small easily handled units, maintaining a good overview of the entire application.

Here is an example demonstrating the power of macro programming:

Example 2.1

```
                .MACRO    BCONOUT dev,char      ;BIOS Function
                clr.w     d0                    ;clear d0
                move.b    \char,d0              ;Put character to
                move.w    d0,-(sp)              ;print onto stack
                move.w    #\dev,-(sp)           ;Device number
                move.w    #3,-(sp)              ;Function number
                trap      #13                   ;BIOS call
                addq.l    #6,sp                 ;correct stack
                rts                             ;return from call
                .ENDM

                .MACRO print string,device      ;Print Macro
                .DATA
                .IF       \?string              ;if string exists
\~a:            .DC.b     \string               ;in datasegment
\~b :                                           ;pass to output
                .ELSE                           ;otherwise
\~a :           .DC.b     13,10                 ;pass CR
\~b :
                .ENDIF
                .TEXT
                lea.l     \~a,a3                ;string address
                moveq.l   #\~b-\~a-1,d3         ;string length -1
.\~:            BCONOUT   \device,(a3)+         ;print 1 char
                dbra      d3,.\~                ;until all chars
                .ENDM                           ;printed

macro_call:
        print 'Hello!',2
        print ,2
```

2.1.2.1 Macro Definition

.MACRO Macro_name {name_1}{,}{name_2}

or

.MACRO_Name.**MACRO**{name_1}{,}{name_2}

A macro definition is established through the Pseudo Opcode '.MACRO'. The pseudo opcode is followed by up to 16 parameter names. Each parameter name consists of up to 16 characters. These parameters may not be interchanged with Labels. They don't define any symbols, but they permit the symbolic testing of data passed by calling a Macro. For this reason no Label operators can be used. The text following .MACRO makes up the body of the Macro. There are absolutely no limitations placed on the commands used within them. A macro definition is ended with the '.ENDM' directive. A macro can be exited early using the '.EXITM' instruction.

Example: 2.1:

```
.MACRO Macro_name Parameter
;
;what is to be executed by the assembler when this macro is
;called is found here.
;
.ENDM
```

An exception to this is calling a macro from within a macro. Only the inner most .MACRO .ENDM Command is assigned to this macro definition. We do not recommend the use of this variation.

2.1.2.2 Macro Call

Macro_name {Parameter_1}{,}{Parameter_2}...

A macro can be called after it has been defined. A macro call allows the assembler to execute the instructions defined in the body of the macro at the current point int the source code. Any number of macro's may be called from within a macro (see Example 2.0). The parameters following the call name of the macro can be any expression that is recognized as syntactically correct by the editor including address types. We recommend studying the included operating system definitions to build a better understanding of the implementation possibilities of macro programming.

2.1.2.3 Special Macro Functions

There are many special functions possible within source code by using a macro.

Checking the parameter passed:

\ Parameter

The 'Backslash' has the following name.

\\Parameter

If one of the corresponding Macro definitions in 'MACRO Command' matches the parameters specified, the expression will be calculated and placed in the macro at the position of the given parameter instead of checking the parameter.

Note:

a) There may be many such parameters passed in expressions calculated together. Because the syntax pays no attention to later calls during Macro definition, the programmer must ensure that different addressing types are used or combined with each other.

Example 2.3.1.1: Correct

```
.MACRO Example_1 Parameter_1,Parameter_2
move.w \Parameter_1+2(a0,d0.l),\Parameter_2
.ENDM

Example_1 4,(a1)+          ;Call
```

would result in: move.w 6(a0,d0.l),(a1)+

Example 2.3.1.2: Incorrect!

```
.MACRO Example_2 Parameter
move.w #\Parameter,d0
move.w d1,\Parameter
.ENDM

Example_2 #8       ;Call
```

would result in:

```
move ##8,d0        ;Incorrect because of double addressing
move d1,#8         ;Incorrect because of illegal addressing
```

It is important to pay attention to the manner in which macro parameters are used when they are passed to the macro. Errors are caught with the appropriate message.

b) The programmer is not required to set all parameters that were defined during macro definition. More expressions then were defined during macro definition may not be passed. Attempting to do so could possibly lead to a loss of data.

Example 2.3.1.3:

```
.MACRO Example_3 Parm_1,Parm_2,Parm_3,Parm_4
;Body of macro
.ENDM

Call:
Example_3 ABC,DEF,GHI,JKL ;all parameters are set
Example_3 ,,GHI ;only Parm_3 is set
Example_3 ABC,,JKL ;only Parm_1 and Parm_4 are set
Example_3 ABC,,GHI,JKL,MNO ;too many parameters
```

c) Attention must be paid to the search criteria in connection with the framing of Macro calls (Example 2.0). The name used must be exclusive to that macro.

If a Macro_A is called from Macro_B, and Macro_B passes a parameter, Parm_1, this parameter can be reached by either macro with the name given in the definition specified by Macro_b.

Example 2.3.1.4:

Defined:
```
.MACRO Macro_a Parm_2
move.w \Parm_1,\Parm_2
.ENDM
```

as well as:
```
.MACRO Macro_b Parm_1, Parm_3
;...
Macro_A Parm_3
;...
.ENDM

Call from Macro_B:  Macro_B d0,d1
returns: ...
move.w d0,d1
;...
```

'd0' is connected directly to the passing parameter from Macro_Bin Macro_A. 'd1' is first passed from Macro_b in Parm_2

If a parameter has the same name in both Macro_A and Macro_B then the innermost macro (here Macro_A) can only access the value within the macro.

To summarize: If a passing parameter is required, the assembler will first look within the innermost macro (here Macro_A). If it doesn't find the name,it will search in the next outer macro (here Macro_B) until it finds either the normal source code or the name it was looking for.

Notes: Diverse assemblers support the passing of expressions, commands, and address type fragments to macros. Because these assemblers do not work with a tokenized source file format, new commands, symbols, address types, and so forth, can be created using simple text copying.

This is not supported by the GFA ASSEMBLER in order to avoid undermining the intelligent input and structuring principles.

\n

- n may take the values from 0 to 16 and must follow the backslash. The values 0 to 10 have the same effect.

- Parameters passed to macros can be refered to not only by their name, but also by their position in the list of defined parameters. In this way it is possible to avoid the automatic search for parameters with identical names, including those in any higher nesting levels.

Example 2.3.2:

```
.MACRO Compare Source,Destination
;... cmp.b \1,\2
seq Flag
;...
.ENDM
Compare (a0)+,d0
results: ;...
cmp.b (a0)+,d0
;...
```

\#

Returns the amount of the actual parameters passed on to a macro. Can be used as a numerical expression in all calculations.

\?Parameter

Finds out whether or not a certain parameter was passed on during the call.

Returns: true(=–1), when parameter is used false(=0), when parameter is unused

\?n

– n may take the values from 0 to 16 and must follow the backslash. The values 0 to 10 have the same effect.

– Similar to:\?Parameter; it is only used for checking the parameter number instead of the parameter name. This way you can also check values not containing names.

\!Parameters

Finds out the addressing mode of code passed on to a printout. The returned values stand for the following types:

Addressing modes:

1:	Dn	6:	d(an)	12:	#immediate data
2:	An	7:	d(An,Xm)	13:	SR
3:	(An)	9:	absolute	14:	CCR
4:	(An)+	10:	d(PC)	15:	USP
5:	-(An)	11:	d(PC,Xm)		

-1: parameter not passed on
8: not yet used

\!n - n may take the values from 0 to 16 and must follow the backslash. The values 0 to 10 have the same effect.

- compare with \!Parameter

\'Parameter

Checking the type of data.

Returned Value:

-1:	parameter not passed on
0:	parameter is a numerical value
>0:	parameter is a string, the length of the string is returned in bytes

\'n

- n may have the values 0 to 16 and must follow the backslash. The values 0 to 10 have the same effect.

- compare with \'Parameter

.\~{Name} – Macro–specific label definition.

If the user wants to call a certain macro in a program a few times it might happen that redefinition errors occur when symbols are defined in macros. To get round this simply change the symbol name before every call.

The expression responds like a symbol (the second version is local).

While assembling, the '\~' is changed to "M"+Str$ (running macro call number){+"Name"}.

Therefore it does not matter if one refers to the symbol over the previous operator or with the explicit name. (Then one must find out the call numbers by counting the macro calls.)

For more details with the use of symbols please read the corresponding chapter !

Remarks: – From outside a macro you can only reach it's '\~' symbols through 'MxxName'. (The call number has changed again then!)

– If a '\~' symbol is defined outside any macro on the level of a source code it will receive the form M0{Name}.

– The entire length of a '\~' symbol including the 'Mxx' prefix may not be longer than 16 characters.

– Keep example 2.0 in mind !!!

2.1.2.4.1 Combining Source Files

.INCLUDE 'Filename.ext'

.PATH 'pathname' specification of an access path.

If source code is loaded using the .Include instruction, it is used exactly as if it were located at the .INCLUDE Instruction in the main source code. (.INCLUDE itself can be used anywhere in the source text.) If a file with the same name is already in memory (any number of files can be in memory), the file in memory will be used. If this is not the case then the current access path will be searched. Up to 16 File Paths may be specified in which to search for the specified file. The paths are searched in the order in which they are specified, the first file with a matching name will be used.

2.1.2.4.2 Straight text assembly

Macro functions

The tokenised entry with syntax checking by the editor forces the use of syntactically correct and complete expressions. However, it tends to put too many restrictions on the possibilities of programming macros.

Examples:

(a) The following construction would be impossible:

```
        .MACRO Loop Register_1,Register_2,Register_3
.\~Loop:  move.b    (\Register_1)+,(Register_2)+
          dbra      \Register_3,.\~Loop
          .EMDM
```

The call 'Loop a0,a1,d0' was supposed to produce:

```
.M1Loop:  move.b    (a0)+,(a1)+
          dbra      d0,.M1Loop
```

However, the second line of the macro definition would have been rejected when entered.

(b) Composite instructions would not be possible:

```
.MACRO Move extension, source, destination
move.\Extension \source,\destination
.ENDM
```

The editor would not accept 'move.\Extension'.

In order not to impose such restrictions on the user, but also not to do away with syntax checking, the following conventions have been adopted in the GFA Assembler:

All expressions, addressing methods and instruction lines which are to be composed of different fragments and will appear syntactically incorrect to the editor at the time of their entry, must be enclosed in curly brackets {}.

Within curly brackets any number of expressions (including strings and string variables) or addressing methods may be contained, separated from each other by commas or spaces. They will be syntax checked individually, but not as a whole.

During assembly, the individual fragments will be concatenated and interpreted. Strings will be processed without quotation marks. String variables will have their contents passed. Macro parameters will be processed into the expressions passed to them.

Some illustrative examples and notes:

(1) Example (a) must be written like this in the GFA Assembler:

```
            .MACRO Loop Register_1,Register_2,Register_3
.\~Loop     move.b      {'(',\Register_1,')+'},{'(',\Register_2,')+'}
            dbra        \Register_3,.\~Loop
            .ENDM
```

(2) Example (b) in the correct form:

```
            .MACRO Shift Extension Source, destination
            {'move.',\Extension,\Source,',',\Destination}
            .ENDM
```

(3) Expressions passed as strings must only be in curly brackets if they do not pass a syntax check individually:

```
{a,b,c,d,e} and {'a','b','c','de'} both result in
abcde

move.w {a,b,c,d,e},d0 results in
move.w abcde,d0
```

(4) If the quotes used as string delimiters are explicitly required in expressions developed from plain text assembly, they need to be entered twice since strings are processed without quotes:

Example:

The instruction cmp.b #'A',d0 is to be composed:

```
{'cmp.b '#'A','',d0}
```

(5) Curly brackets may also be nested, that is, expression in curly brackets may themselves contain curly brackets. Care needs to be exercised here with strings:

```
{''d0''} produces 'd0'
{{''d0''} produces {'d0'} first, but then d0!
```

(6) Curly brackets may also be involved in calculations and addresses:

```
.MACRO Miracle digit      | produces:
move,w #1+{1,\digit},d0 | move.w #1+15,d0
.ENDM                     |
Miracle 5
```

(7) Generally two basic types of application need to be distinguished:

- {} within expressions and address types (cf. example (1))
- {} as complete command lines (opcodes with parameters)
  (cf. example (2))

(8) Curly brackets must only be used in macros.

(9) Caution: Intensive use of plain text assembly slows down assembly speed significantly.

Label definition with macro parameters

Within macros it is possible to define symbols the names of which can be passed as macro parameters:

Example:

```
                .MACRO Label_definition Name_1, Name_2
\Name_1:        nop         ; \equivalent
\\Name_2:       nop         ; /formulations
                .ENDM
                Label_definition Symbol_1,Symbol_2
```

results in:

```
Symbol_1:nop
Symbol_2:nop
```

However, only parameters which have been assigned a name during the definition of the macro may be passed.

Definitions such as \1: are not possible.

2.1.2.5 String processing

String variables

String variables are familiar from most high-level programming languages. In assembly programming they permit, in conjunction with macros and the new {} functions, an effective parameterisation of source texts.

Five points are basically sufficient to explain their use:

Definition and use are as in absolute symbols defined with .SET.

Strings variables are in all respects equivalent to strings explicitly delimited by quotes. They can be used wherever explicit strings could also be used.

String variables are recognised only during the assembly process, and cannot therefore be imported or exported during linking.

To distinguish them from ordinary numerical symbols, they begin with a colon (:).

Strings can have a maximum length of 255 characters.

Definition syntax:

:Name Stringexpression_1, Stringexpression_2 ...

All subsequent strings are stored (concatenated to one single string) under :name.

Examples:

(a) :String_1 'is'
:String_1 'This ',:String_1,' a sample string'

.DC.b :String_2 is then equivalent to
.DC.b 'This is a sample string'

(b) :Character 'A'
=> cmp.b #:Character,d0 is equivalent to cmp.b #'A',d0

String variables can also be used within curly brackets. It must be emphasised that in that case not the variable itself, but its contents will be read!

String functions

To simplify the use of string variables, the following functions which should be familiar from BASIC, for example – are available:

SYNTAX:	Description:	Example:
mid{string,x,y}	(1)	
	Fetch y characters from string starting with xth character	mid{'123456',2,3} produces '234'
left{string,x}	(1)	
	Fetch x characters from string, starting from left	left{'123456',4} produces '1234'
right{string,x}	(1)	
	Fetch x characters from string, starting from right	right{'123456',4} produces '3456'
dup{string,n}	(1)	
	dupilcate string n times to form new string	dup{'123',4} produces '123123123123'
len{string_1,..,string_n}	(2)	
	gives the total length of the specified strings	len{'123','abc'}. produces 6
asc{string}	(2)	
	gives the ASCII value of the first character of the string	asc{'ABC'} produces 65
chr{expression}	(1)	
	produces one-character string of ASCII value expression	chr{65} produces 'A'
instr{string_1,string_2}	(2)	
	searches string_1 for string_2 Returns: -0_not found -n,n >0 string_2 starts from nth character in string_1	instr{'123456','345'} returns 3

(1) Result is string
(2) Result is numerical value

If the result of a string function is again a string, the function may be used like a string variable wherever strings are permitted.

Caution:

The curly brackets used with string functions should not be confused with those used in plain text assembly.

2.1.2.6 The ILLEGAL instruction

The assembler treats the 'illegal' instruction as a macro. This is not a bug but intentional in order to permit the user to freely select an illegal opcode. Should the 'illegal' instruction be required, therefore, it can easily be defined by the user:

```
.MACRO illegal
.DC.w $4afc ; or another illegal code
.ENDM
```

2.1.2.7 Including additional source text

```
.INCLUDE 'FILENAME.EXT'
.PATH 'Filepath'    ; specification of search path
.INCDIR 'Filepath'  ; synonymous to .Path
```

Source text loaded subsequently with the INCLUDE instruction will be processed by the assembler in the same way as if it was found directly in the current source. (.INCLUDE may be used anywhere in the source text).

If a text by the same name is already in memory (since you can have as many texts in RAM as you like!), that text will be used. If this is not the case, the current path will be searched for it.

Additionally, the user can specify up to 16 file paths to be searched for the text. The paths will be searched in the order in which they were entered; the first file found with the searched-for name will be used.

2.1.2.8 Additional macro functions from Version 1.3

To make it easier to port source code into the GFA ASSEMBLER format, the following has been added to the existing methods of macro expansion (with parameters):

Lines beginning with an '@' character will, like pure comment lines, not be tokenised when entered. Only during assembly will the line be processed as follows:

The line will be searched for the '\' backslash character, to find macro functions.

If parameters are passed to the line, they will be inserted in the position of their placemarkers (\parameter_name).

The resulting line will be tokenised and assembled.

Advantages of this procedure:

Easier to import alien formats, since macro expansion works in the same fashion (simply prefix line with '@'!).

Complex macro functions such as the copying together of new instructions and addressing methods without recourse to the GFA ASSEMBLER syntax with curly brackets.

Disadvantages of the procedure:

No syntax checking during input.

Slow assembly speed, so that intensive use over entire program sections is not advised.

Example: Generating an addressing method

```
Existing GFA syntax                          |New method from Version 1.3
---------------------------                  |--------------------
.MACRO example_old addressregister           |.MACRO example_new
                                             |addressreg
move.l {'(',\addressregister,')'},d0         |@move.1 (\addressreg),d0
.ENDM                                        |.ENDM

      Example a0                             |Example a0
```

All functions with curly brackets are only permitted within macros, with the exception of the w{} and l{} functions described further below.

As from Version 1.3, compliance with this syntax is checked and any deviation produces an error message.

If the user wishes to use the {} functions outside his currently defined macro structures, the following trick is recommended:

```
; put includes, (macro) definitions etc here
;
.MACRO main
;
; The program with the required macro functions
;
.ENDM
main  ; call macro
```

The above structure is therefore broadly comparable to the C function main{}. The entire program code is nested within a macro, which becomes a program when called at the end of the source text.

dec{expression} is equivalent to the BASIC function STR$(exp.n)
hex{expression} is equivalent to the BASIC function HEX$(exp.n)
bin{expression} is equivalent to the BASIC function BIN$(expression)
oct{expression} is equivalent to the BASIC function OCT$(expression)

The four functions above are used to transform numerical expressions into strings. The resulting character strings may be used in macros wherever explicit strings in the format 'string' (text enclosed in quotes) would also be permissible.

Examples:

```
.DC.b dec{35} is equivalent to .DC.b '35'
.DC.b hex{35} is equivalent to .DC.b '$23'
.DC.b bin{35} is equivalent to .DC.b '%100011'
.DC.b oct{35} is equivalent to .DC.b '@43'
```

NARG resp. narg

Yields the number of the parameters passed to a macro as a numerical value. The effect of this function is completely identical to \#. It was include to enhance compatibility with other assemblers.

^^@

Returns the call number of a macro. Whenever a macro is called, the assembler increments a counter. The current value of the counter can be interrogated with this function. Thus you can, for example, implement symbol definitions which change from macro call to macro call.

Example:

The symbol definition	\~SymbolRest:
when using the ^^@ function reads	{'M',dec{^^@},'SymbolRest:'}

The following functions are permitted everywhere, including outside macros:

w{address} forces absolute short address
l{address} forces absolute long address

The following restrictions apply to 'address':

'Address' must not be relocatable.

'Address' must be located in the first or the last 32k block of the address area and thus be representable as a word. (The processor expands this to a longword when addressing).

Non-compliance with these points will produce an appropriate error message.

2.1.2.9 Including absolute data

.IBYTES 'DATANAME.EXT'

The file specified with 'DATANAME.EXT' is inserted at the the current program counter position as a data packet. The data contained in the file will not be altered by this operation. This instruction enables a problem-free transfer of data into the source text. One possible application would be the inclusion of a complete bitmap into the data segment of a game.

The .IBYTES instruction, like .INCLUDE, observes the search paths specified with .PATH.

2.1.2.10 Using symbol definition files

```
.ITABGEN                    ;Generate symbol definition file
.ITABEQU 'DEFNAME.EXT'      ;Call symbol definition file,
                                .EQU-definitions
.ITABSET 'DEFNAME.EXT'      ;Call symbol definition
                                table, .SET definitions
```

The instructions pair **.ITABGEN** and **.ITABEQU/.ITABSET** permits the generation and use of symbol definition files. When a source text uses numerous constants (that is, absolute and not relocatable symbols), to symbolically address operating system functions or hardware addresses for instance, it is often inconvenient to burden the program text with this or to read in .INCLUDE definition files. The user can use the following approach:

1. Generate a source text containing all definitions of the constants (and only these).

2. Insert the .ITABGEN instruction into this text.

3. Assemble the text. No program or object module will be generated, but a special symbol file with the name of the source text and the extender '.TAB'.

4. Insert the .ITABEQU 'source_text_name', and .ITABSET 'source_text_name' into any position in the source text which is to use the constants.

During assembly all definition instructions from the symbol file will be executed as .EQU definitions where the .ITABEQU, and as .SET definitions where the .ITABSET instruction is encountered.

The method described requires significantly less memory space and processing time than defining the constants in an include file or directly in the source text proper.

A source text with the .ITABGEN instruction will be assembled like an ordinary source text, i.e. macros, conditional assembly and so on are performed to the full extent. The only difference therefore consists in the storage method. Any code or relocatable symbols are ignored during the generation of a .TAB file!

The .ITABEQU and .ITABSET instructions, like .INCLUDE, observe the search paths specified with .PATH.

Structure of the symbol file:

Each entry into the symbol file is 20 characters long. It consists of a longword with the value of the constant to be defined and 16 characters with the symbol name. In the event of symbols with a length of more than 16 characters being found in the .ITABGEN file, the extra characters are cut off. With symbols of less than 16 characters, the free bytes are filled with zeros.

In the last valid symbol bit 7 (%1000 0000) is set.

Example:

```
.ITABGEN file:
Name: 'SYMBOL.TAB'
.ITABGEN
.ABS 0
Name: .DS.b 8
Status: .DS.b 2
Value: .DS.b 4
.END
```

```
| .ITABEQU file:
|Name: any
|.ITABEQU 'SYMBOL.TAB'
|;...source text
|move.w status(a0),d0
|move.l value(a0),d1
|;...
```

2.1.3 Conditional assembly

The main field of application for conditional assembly is in conjunction with macros. Thus a macro can react differently depending on the parameters passed when called, or the assembly of an entire source text can be very flexibly controlled with only a few starting values.

Conditional assembly is very easy to use, because the If-else-endif structure so familiar from many high-level programming languages is used. 'Else' has an alternating effect, i.e. source text named after a double 'else' will be processed again at 'true'(=-1). There is also a one-line variant: .IIF

Conditional assembly may be nested to any depth.

IF...ELSE...ENDIF...

```
;..
.IF Condition
; ...
; assembled if Condition true(=-1)
; ...
.ELSE ; ...
; assembled if Condition false(=0)
; ...
.ELSE
; ...
; assembled if Condition true(=-1)
...etc...
.ENDIF
```

.IIF Condition, Command...

The commands following the condition are only assembled if true (=-1). This structure is ended with the end of the line character.

This is best illustrated with examples:

Example 3.1: GEM DOS Call with Stack Correction

```
.MACRO GEMDOS Trap_Number,Stack_area
move.w      #\Trap_number,-(sp)     ;Gemdos function #
trap        #1                      ;Trap call
.IF    \Stack_area<=8               ;if the trap to correlate is
addq.l      #\Stack_area,sp         ;not longer than 8 bytes
.ELSE                               ;then faster addition
adda.l      #\Stack_area,sp         ;commands can be used.
.ENDIF
.END
```

Example: 3.2 Dump a string and it's address

```
      .MACRO Dump String,Address,Segment
      .TEXT
      .IF \?Address                 ;if dump is possible
      move.l      #\~,\Address      ;for given string address,
      .ELSE                         ;otherwise put address
      pea         \~                ;onto stack, and set the
      .ENDIF                        ;string in the
      .SECTION Segment,DATA         ;given data segment and
\~:   .DC.b \String,0               ;mark it with Symbol
      .TEXT
      .ENDM
```

Example 3.3 Conditional exit from a Macro

```
.MACRO Increment value,Line         ;Macro to increment
.IIF !\?Line,.EXITM                 ;Exit if no line is given
addi.l      #\Value,\Line
.ENDM
```

In addition to these universally known forms for giving the condition input, there are various other command variations in which a condition is tested. The following text is used when the condition is true.

.IFEQ Expression ^...**ELSE** ...**ENDIF** etc..**IIFEQ** Expression , ...

Expression=0.

.IFNE Expression ...**ELSE** ...**ENDIF** etc..**IIFNE** Expression, ...

Expression<>0.

.IFGT Expression ...**ELSE** ...**ENDIF** etc..**IIFGT** Expression, ...

Expression<0.

.IFGE Expression ...**ELSE** ...**ENDIF** etc..**IIFGE** Expression, ...

Expression<=0.

IFLT Expression ...**ELSE** ...**ENDIF** etc..**IIFLT** Expression, ...

Expression>0.

.IFLE Expression ...**ELSE** ...**ENDIF** etc..**IIFLE** Expression, ...

Expression>=0.

Additional synonyms for IF from Version 1.3

.IFD symbol/expression equals	.IF ^^ defined symbol/expression
.INFD symbol/expression equals	.IF ! ^^defined symbol/expression
.IFC string1,string2 equals	.IF ^^ streq string1,string2
.IFNC string1,string2 equals	.IF !^^ streq string1,string2

Example 3.4 Block Move

```
        .MACRO Move_block Start,Finish,Length
        .IFNE Length                        ;Memory moving routine
        lea.l       \Start,a0
        lea.l       \Finish,a1              ;If the block to be
        adda.l      \Length,a0              ;moved is 0 Bytes
        adda.l      \Length,a1              ;in length, break with
        move.w      \Length,d0              ;error message (.FAIL)
        subq.w      #1,d0
.\~:    move.b      (a0)+,(a1)+             ;(copy loop)
        dbra        d0,.\~
        .ELSE
        .FAIL
        .ENDIF
        .ENDM
```

.ASSERT Condition

– Breaks the assembly with an error message if 'Condition' is False, when it has a value if 0.

.FAIL

– Always breaks off assembly. (See Example 3.4)

Notes: Programmers can attempt to use a conditional assembly, to distinguish between the passes of the assembler by calculating symbol differences in optimization mode. This type of destructive programming leads nowhere and ultimately confuses the assembler. It is recognized and forbidden.

2.1.4 Repeated Assembly

Repeated assembly means exactly that, the repeated assembly of specified source file. This technique saves work when entering the source code and increases the readability of the listing. Applications that require extreme speed make use of this technique.

The enhanced memory requirement for the identical Program Code part is exchanged for no loss in time through a loop (in the program). Also, a running variable can be established with a Symbol definition and source code segments with changed values can be repeated.

```
.Syntax:    .REPT Expression
            ;...
            ;The text that is to be re used is found here.
            ;...
            .ENDR
```

The text bracketed by .REPT and .ENDR is directly assembled.

Example: 4.1:

Quick Block erasing for Bit Pattern graphics programming (16 Longwords).

```
Repeat form :     | Alternative:                | Conventional:
                  | (with running var: INDEX)   |
                  |                             |
lea.l Base,a0     | lea.l Base,a0               | lea.l Base,a0
. REPT 16         | INDEX set 0                 | clr.l (a0)+
clr.l (a0)+       | .REPT 16                    | clr.l (a0)+
.ENDR             | clr.l INDEX *4(a0)          | ... repeat 16x
                  | INDEX = INDEX+1             | clr.l (a0)+
                  | .ENDR                       | clr.l (a0)+
```

Example 4.2:

Initializing a Memory Block with numbers from 0–255

```
Loop :    .DATA                  | Conventional: .DATA
          Number = 0             | .DC.b 0,1,2,3,...
          .REPT 256              | ...
          .DC.b Number           | .DC.b ...,253,254,255
          Number = Number+1
          .ENDR
```

Hint:

If a Symbol is defined within a .REPT loop, there may be problems with redefinition. The use of local symbols and the command .LOCAL at the beginning or end of each loop iteration can help with this problem.

2.1.5 Commands for Memory Initialization

2.1.5.1 Text and Initialized Data Segments (.DATA)

There are many varied application possibilities for memory initialization, for example transferring text, jump tables, bit patterns, and so forth, in data segments or dumping special data in text segments or LINEA Opcodes.

.DC{.x} Expression {,Expression}{,Expression}...

.x The working width: .b : Byte, .w : Word, .l : Long word, Expression – any numerical value. The value is inserted at the given object code position (program counter) in the given data width.

If the data width is a byte length, strings are also permitted. Strings are inserted character-wise.

Example: 5.1: Loading a string into memory.

```
.DC.b 'Hello!' is identical to
.DC.b $48,$65,$6c,$6c,$6f,$21
```

.DCB{.x} Number,Fill_Pattern

The .DCB-Command executes the command '.DC{.x} Fill_Pattern', Number times.

Example: 5.2: Initializing a memory block.

```
.DCB.w 5,-1 is identical to
.DC.w -1,-1,-1,-1,-1
```

.INIT{.w} {#Num_1,}Pattern_1{.l}{.Pattern_2{.w}}{,#Num_2,Pattern_3{.b}}...

The .INIT command is the easiest of the commands available for memory initialization. #Num_1 and #Num_2 each contain the number of times the value is to be set with the given data width at the current position. The Default data width is .w (word). The data width default may be changed to the width specified in the .INIT command, after an .INIT command is executed for all additional Fill Patterns. This instruction is especially useful when longer structures with mixed data types and widths must be defined.

Example: 5.3:

```
.INIT.w -1.l,0.w,#8,'String'.b,#3,'A'.b,'B'.b
; is identical to
.DC.l -1
.DC.w 0
.DCB.b 8,'String'
.DC.b 'A','B','A','B','A','B'
```

2.1.5.2 Uninitialized Segments .BSS and .ABS structures

These are explained in more detail in another section of this manual.

.DS{.x} Number

The program counter is incremented and an area of memory is reserved to allow room for the data expressed by Number with the specified data width (byte, word or long word) starting at the current address. This area is then reserved.

Example: 5.4:

```
.DS.w 30 reserves 60 Bytes
.DS.l 2 reserves 2 Bytes
```

.EVEN

The EVEN instruction justifies the current PC address. Fill bytes of 0 are inserted into the text and data segment.

.CNOP offset, modulo

The CNOP instruction ensures that the following program code begins at an address which is a multiple of 'modulo'. Additionally the 'offset' value can be added to this address. To produce an address with the required properties, rounding-up is used. If the CNOP instruction is not used in the BSS segment, the 'empty space' up to the CNOP address will be filled with zeros.

Offset and modulo may be numbers in the range of 0 to 255.

Examples:

```
.CNOP 3,4   ;The following code starts 3 bytes after the next
            ;longword address
.CNOP 0,5   ;Address to be a multiple of 5.
.CNOP 0,2   ;equivalent to .EVEN
```

2.1.6 Commands for Formatting Listings

During debugging of a program it is often helpful to have the action of the assembler recorded either entirely or in part. This supplies the programmer with a detailed picture of critical program sections, and with intensive macro programming, a complete listing of the code created for debugging. An assembly record can be sent to the screen, an attached printer, or a file (on a floppy, hard disk, or RAM disk.) These files use the .LST extension. By using special directives in the source code, the user can specify areas of special interest for recording.

.LIST

Activates the recording process. All work done by the assembler following this command is shown according to the format as illustrated in Example 6.1.

.NOLIST

Turns off the recording.

.TTL 'Header line'

.SUBTTL 'Footer line'

Each recorded page is organized as follows:

```
System title line (Action,File,Date,Page)
{Header Line} (Definable by the user)
----------Empty line----------
....
Listing
....
----------Empty line----------
{Footer Line} (Definable by the user)
```

The definition of the Header and Footer line is used for all such lines following it's definition. After the definition of the first Header and Footer line all following definitions will initiate a form feed.

.PLEN {expression}

Sets the page length of a recorded page.

.LLEN {expression}

Sets the line length.

Page and line length can only be changed when the recording is inactive. (For example, before a .LIST or after a .NOLIST instruction)

Hint:

Page Format Specifications:

- Five lines of each recording page are reserved for Header and Footer lines. All remaining lines are used for the listing output. If no Header Footer line is specified, an empty line will be used in it's place. (Compare to .TTL, SUBTTL).

- If no page format is specified by the user in the source code, the assembler will use the information found in the corresponding Printer Parameter Screen (Menu 5,1) (For output through a file or printer). If a Header line is active in this menu, the number of lines available is reduced by 3. The actual number of characters per line (columns) can be obtained after searching the left edge and eventually the line numbering (compare to the Editor). If this data is less than the minimum value of 80 characters per line and 6 lines per page , a default format of 80 characters on 58 lines will be used. If the output is to the screen, the page format is always 80 characters on 25 lines (one screen page). After each page is output a key must be pressed to go to the next page. This permits browsing through the output.

– With printer output, each page is sent to a page buffer before it is output. The size of this buffer may be specified when the printer is set up in the Printer menu of the Editor. It is configured as follows:

Columns*(lines–Header line)*(Characters+edge)

with:	Columns:	number of Columns
	Lines:	lines per page
	Header line:	0 with no Header line
		3 with a header–line
	edge:	Characters per column

The assembler may not exceed this buffer size with a changed page format. If this is the case each line on the page will be shortened so that it will fit on the page in the buffer. If there is enough room available, it is recommended that a large buffer be installed for recording in order to avoid long delays during assembly.

– In principle, it is possible to use many columns for assembly recording. However, because each line must be at least 80 characters wide, this will depend on the printer used.

– For printing, the parameters set during the printer set up in the Editor will usually apply.

– Please do not mistake the header line of the record with the header line of the Printer menu. The header line of the Printer menu will be suppressed during the process of printing the record.

.LMODE Expression 1 <=Expression <=3!

As illustrated by example 6.1, an assembly recording is formatted in column orientation. If the contents of a column exceed the reserved length, it can be dealt with as follows:

1. Place output into the next line at the corresponding column position.

2. Cut off the line.

3. Continue writing on the next line. All additional columns will be moved to the right.

.PRINTER Expression {,Expression} ...

Like the .DC.b command strings can be specified here. If a record is being output to the printer at the time the given character will be sent to the printer and ended with a carriage return, line feed. This is used to send printer commands, for compressed print mode, other fonts, etc.

Example:

.PRINTER 27,15 Compressed fonts.

.PAGE{expression}

This command, without any parameters, sends a form feed. When a parameters is passed, sends a form feed if there are fewer lines available then expression (not counting lines reserved for Footer lines).

.SPACE{expression}

Inserts expression empty lines. If no parameters is given the, default value of 1 will be used.

.MLIST .NOMLIST

.MLIST Macro calls are listed in expanded format.

.NOMLIST Only macro calls are listed, not the contents of the macro itself.

.CLIST .NOCLIST

.CLIST The sections that are jumped over and not assembled are listed during conditional assembly.

.NOCLIST The actual instructions assembled in an .IF structure are listed.

Example: 6.1

The following Assembler record excerpts are from the assembly of the source code in Example 2.0. A format of 80 characters by 20 lines is used.

Column 1

Contains the line number of the work step in the source code. This is concluded with one of the following status characters:

. The line is from a macro call
: The line is from an .Include File
+ During conditional assembly : true
– During conditional assembly : false

Column 2 Shows the running Object code address (referred to as Base address 0). Be careful about changing Segment types. Programs begin the data segment at the end of the text segment. .BSS segments begin at the end of the data segment. In Object modules all segments have a base address of 0.

Column 3 Gives the assembler code created in Hexidecimal format.

Column 4 Reserved for symbol definition.

Column 5 Lists the executed steps in clear text.

Column 6 Contains any comments.

```
{System-Status-Row: Explanation                                    }
{Version:    Function:    Filename    Date: Time: Page:           }

GFA Asm V1.0 Asm-Prot.:D:\PRINT.LST Day:6.2.1990 Time:2:9:0 P:1
This is the headline.

6       000000                      .TEXT
8       000000                      .MACRO bconout dev,char ;BIOS
18      000000                      .MACRO Print String,Device ;Macro
34      000000                      .LMODE 2
35      000000                      Print 'Hello!',2              ;print
19      000000                      .DATA
20 .    00003c                      .IF \?String                  ;if th
21 .+   00003c 48616c6c6f21\~a      .DC.b \String                 ;in Da
23 .+   000042                \~b   .ELSE
24 .-   000042                \~a   .DC.b 13,10                   ;other
26 .    000042                \~b   .ENDIF
Here goes the footer.
{form feed}
GFA Asm V1.0 Asm-Prot.:D:\PRINT.LST Day:6.2.1990 Time:2:9:0 P:2
This is the headline.
28 .    000042                      .TEXT
29 .    000000 47f9 0000003c        lea.l \~a,a3                  ;Strin
30 .    000006  7605                moveq.l #\~b-\~a-1,d3         ;Strin
31 .    000008            .\~       bconout \Device,(a3)+ 1       ;Tim
9 .     000008  4240                clr.w d0                      ;Bcono
10 .    00000a  101b                move.b \char,d0               ;out t
11 .    00000c 3f00                 move.w d0,-(a7)               ;onto
12 .    00000e 3f3c 0002            move.w #\dev,-(a7)            ;Outpu
13 .    000012 3f3c 0003            move.w #3,-(a7)               ;Funct
14 .    000016 4e4d                 trap #13                      ;BIOS-
15 .    000018 5c8f                 addq.l #6,a7                  ;Stack
16 .    00001a                      .ENDM
32 .    00001a 51cb ffec            dbra d3,.\~                   ;loop
33 .    00001e                      .ENDM
36      00001e                      .SPC 3
Here goes the footer.
{form feed}
```

Recognised, but ineffective instructions:

(The instructions described below are recognised and tokenised by the editor, but will be ignored by the assembler. They have been implemented for reasons of upward compatibility to future versions.)

.IDNT 'string'

.AMIGA

2.2 Calculations, Symbols, Operators

2.2.1 Definition of a Symbol

The assembler distinguishes between relocatable and absolute symbols.

Relocatable symbols are symbols that are at a position relative to the program start. Absolute symbols can be any given value. They are comparable to 32 Bit Integer_variables in BASIC. When a symbol is defined, the type is specified next to the value, and affiliation to a segment is reported. Symbols may be up to 127 characters long, but only the first 16 characters are significant. They can basically be defined in two ways:

– By marking symbols in the program text. This type of symbol is always relocatable. Offsets to the start of the program or segment are used.

– By direct assignment with the following syntax:

Symbol_name {assignment–command} Value

Assignment commands are:		equ, .equ , == (one shot definitions)
	as well as:	set, .set, .= (repeatable definitions)
	and:	reg, .reg (register listing definition.)

The symbols so defined carry the assigned expression as an attribute.

Example:

Symbol equ Symbol_2
If Symbol_2 is relocatable then Symbol will also be relocatable.

A Symbol defined as a label can have two meanings.

- Local Symbols have a period before the symbol name and are only recognizable between two not local symbols. Because of this, the programmer may have to tie together many local symbol names in an especially long source listing. (Local areas can also be separated by using the directive .LOCAL without a non local label). In the assembler, local symbols can also be exported.

- Normal symbols are known throughout the program. Jump points in subroutines are marked with normal symbols, symbols within the subroutine are locally defined and when necessary separated by using .LOCAL.

- Global symbols are exported. Global declaration is only necessary when working with Object Files. They can also be used in other ways.

- During the assembly process, settings can be adjusted for attaching a Symbol table if all symbols or only global symbols are to be included in this table. In order to sue a symbol table for debugging, define all symbols as global.

- Global symbols can also be defined by using the '::' combination in addition to the corresponding Pseudo Opcodes.

Example:

```
Label_1:    ;Label_1 is normally defined
Label_2::   ;Label_2 is globally defined
```

2.2.2 Retrieve Symbols

Symbols may appear at any position in the source code where a numerical value would normally be permitted.

2.2.3 Calculations

Calculations are made in GFA ASSEMBLER according to a specific hierarchy and with as many brackets as desired. Each numeric expression may be the result of a calculation. The following operations are declared:

Logical comparisons with boolean results (0 and –1) (lowest priority):

```
=     : Equal to (logical)
<>    : Not equal
<     : Less than
>     : Greater than
<=    :Less than or equal to
>=    : Greater than or equal to
```

Arithmetic operators (sorted in ascending priority):

```
+     : Addition in 32 bit words
–     : Subtraction in 32 bit words
*     : Multiplication of 16 bit word
/     : Divides a 32 bit word by a 16 bit word
      : / the sign is affected!
&     : 32 bit AND operation
|     : 32 bit OR operation
^     : 32 bit EOR operation
%     : Modulo
<<    : Bit Shift left
>>    : Bit Shift right
```

Further operations: (expressions are presented):

```
– negative prefix (2's compliment)
~ bitwise NOT (1's compliment)
! logical NOT (result is 0 or –1)
```

Special functions:

*	: current PC
^^defined symbol	: is Symbol defined? *)
^^referenced symbol	: Is Symbol referenced? *)
^^Streq string1,string2	: String comparison
^^date	: Current system date in GEMDOS format
^^time	: Current system time in GEMDOS format
^^macdef Macroname	: Is macro defined?

*)Note:

With a multipass assembler, care must be taken to ensure that each pass through the source code reaches the same conclusion with each .IF Pseudo Opcode. For this reason, only those symbols that have been encountered before ^^defined will be considered as having been defined. The ^^referenced function has therefore only limited application. It is included in the command set only to furnish compatibility with other assemblers.

If calculations other than + and - are executed the symbol value is absolute.

If only + and - are used with odd numbers from relocatable symbols, the result will also be relocated.

3.0 Linker

The Linker plays an important part when used in combination with a higher programming language and in the development of large programs. Object modules of the same format can be combined to form new programs regardless of when they were written. Difficult functions can be used from Routine Libraries (for example, from a development package). The programmer must not reinvent the wheel with each new program.

The Linker format used by the GFA ASSEMBLER is the same as that used with Digital Research C (supplied with the Atari ST Developers Package). This format has some restrictions that a programmer must know: No calculations can be made during the linking process. This means that during object module creation, the most that can be done is that expressions can be imported. Values derived from symbols can not be changed during linking.

– The distinctions between the assembler segments can not be done by the linker. Only the segments TEXT, DATA and BSS are created (see above) by the linker. During linking, the message and warning windows from the assembler are displayed.

The user will be able to see what the Linker is actually doing as it ties in various modules from the library. If assembly takes place during the linking, any messages from the assembler will appear in the Linker window.

To differentiate between messages from the Assembler and the Linker, the following special characters are used:

>>, >	followed by assembler messages
= , =	followed by linker messages
!	followed by prompts to the user
–>	followed by display of processing time of assembler/linker

In order to automate the Assembler Linker process and reduce the of commands to be entered, the linker can be controlled by a linker control file. This is a normal ASCII File which must be created in the source text mode of the Editor. Only one command per line is permitted.

Operation

The following steps are required to start a linking operation:

- Creation of an ASCII text in the editor's "General source" mode as a link control file under any filename. This text must contain the linker commands described in the manual.

- All editor functions can be applied to link control files, including Save, Load etc.

- To start the Linker, select menu 6,4. To be on the safe side, the Link menu will only be displayed if an ASCII text (not tokenised) is in memory. The switches shown in that menu have the following purposes:

"generate"

Program: Linker output is a program.

Object code: Linker output is again an object module. This is equivalent to the "-p" control file command (partial linking).

IMG file: Supervision of relocations (see manual).

Absolute: Activates the "-a" command. Important: The "-a" command is only executed if this button is also active. It is thus possible to generate an executable program as well as absolute code without amending the control file.

"Symbol table:" cf. "-l" and "-s" commands in the link control file

"Load map:" cf. "-m" command in link control file.

"Link" starts linking.

Command: (path_name is a place holder for any file path)

path_name Include an object module.

?path_name Include a library (archive).

When linking a library of individual modules, the linker will determine which references have not yet appeared (that is, which symbols are imported without being exported, and so are not defined).

Finally, all the modules of the library are searched for export of the missing symbols. If the linker is successful the exporting module is included.

Hints:

The analysis of an extensive library for internal connections during linking is too time consuming. For this reason a library is used in combination with it's 'Indexfile'. A large amount of information is used by the linker for handling the corresponding library. The first time a library is used an index file is created and saved using the filename extension .NDX. This file is then ready for future reference. This file can also be created using the Library Management functions. More information about the construction and function of the index file is in the appendix.

If a symbol is exported from many modules, the symbol from the last module will be used.

When exporting many individual modules (not belonging to a library) containing a symbol with the same name, the first of those exported will be passed on to other modules.

@path_name

Combines all modules of a library (in alphabetical order).

#path_name

Includes a Binary File. The specific file is placed in a DATA Segment. If you want to be able to reference the data by using a symbol, we recommend that a symbol be defined at the end of the data segment of the object module.

Application example.: passing text, bit pattern graphics,resources, etc.

^Path_name

The specified file will be assembled during linking. There will be no output to the disk. The object code will be directly linked to the modules in memory. The primary advantage of this is that no additional access to the disk is necessary. This option can be executed only once during linking. The file to be assembled must be in memory.

This option is most useful when only one of several object modules has been modified, and the programmer wants to update the current program.

WARNING: The '^' option is not compatible with library manage-ment. This means that symbols using '^' can not be created using a library.

(Except, of course, if they are also required elsewhere and have therefore already been imported). If this is a problem, the user should use the '=' option already described above,to which this restriction does not apply.

=Path_name

Assembly during linking. If a file with the specified name is already in memory, it will be assembled. Otherwise the file will be loaded from the specified path. After assembly, an object file will be written to disk and will be available for further link executions.

The file will also be inserted during the current link process.

As many such assemblies as required can be carried out. The parameters set in the Assembler Menu are used during the assembly.

The parameter settings made in the Call Assembler menu (menu 6,2) apply to the assembly.

Command Operating parameter setting. The hyphen follows one or more of the following commands:

s or S – Add a symbol table,only for Export and Import

l or L – Symbol table with all symbols

m or M– Load map. An alphabetical directory of all symbols to be used.

p or P – Partial linking: create an object module instead of a program.

u or U – 'Unresolved'. Object modules with unused references would normally lead to break in the linker. This command will permit the linker to continueexecution. The linker uses 0 for the error filledsymbols.

a or A – Absolute linking. Separated from segment specific code by a specified address instead ofrelocatable code. This could be used for codeintended to be burned into an Eprom.

Syntax: Parameter_1(text) Parameter_2(data)Parameter_3(bss)
The parameters can use the following format:
r The segment is relocatable.
x Exactly the same as the previous segment.
(Identical to 'r' for parameters with text).
$xx (Hex number) The segment encounteredshould be relocated to the address specified by the Hex Number.

Hints:

Data created by absolute linking has a different format the header. The following construction applies:

Offset:	Contents:
$00	Length of the text segment (a
$04	Length of the data segment
$08	Length of the Bss segment
$0C	Length of the symbol table
$10	Base address of the text segment (b
$14	Base address of the data segment (b
$18	Base address of the Bss segment (b
$1C	End of header

(a = Bit 31 is set if relocation information is present.
(b = Bit 31 is equal to 0, when the data is relocated.

The operating system will not support the absolute linked file format. The programmer must ensure that each segment in memory is correctly positioned.

The 'A' Command following other commands in the same line is not evaluated.

+ Comment line
; Comment at the end of the line.

A comment can be inserted following '+' and ';'.

4.0 Library Management

If you'd like to create your own libraries, you will need a utility for library management. Such a utility is integrated into the GFA ASSEMBLER in the 'Archive ...' menu. It may be used similarly to the way file management is handled under GEM. When the user selects the 'Archive ...' menu item under the Assembler Menu header, the file selection screen will appear and the user can select the desired library. If a name is given that is not available, or does not belong to a library, a new library with this name will be created. Otherwise the desired library file will be loaded. This work screen will now appear:

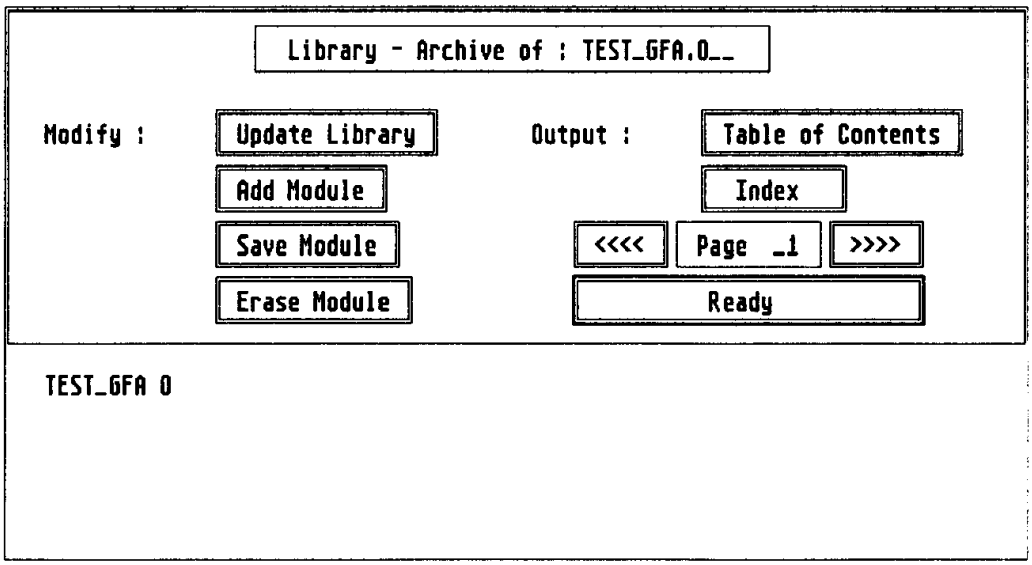


Figure 15 Library Screen

The contents of the library will be shown in the lower part of the screen. The work functions are executed in the upper part of the screen.

<< or >> Page through the contents of modules.

Update Library

The date each module was established is noted, as in the directory under GEM. If the user changes the library, a new date will be recorded. This option will automatically update the library. Individual object modules which were changed are searched for in the folder from which the library was loaded. If the date and time is older than the new object file, it will be replaced by the new file. If this function is selected, make sure the current system time is correct in order to avoid destroying valuable information through an incorrect date and time setting.

Add Module

Entire libraries or individual object modules can be inserted into a library to expand it. They are selected through the fileselect screen as needed.

Hints:

A library is a collection of separate object modules into one file. No sub-directories are supported by the file management.

Each module in a library must have a specific name. (Otherwise it would not be possible to distinguish between modules.) A box will appear in which names can be changed. The affected file can be overwritten, or the process can be stopped if the module name already exists.

A library can contain as many modules as memory will permit.

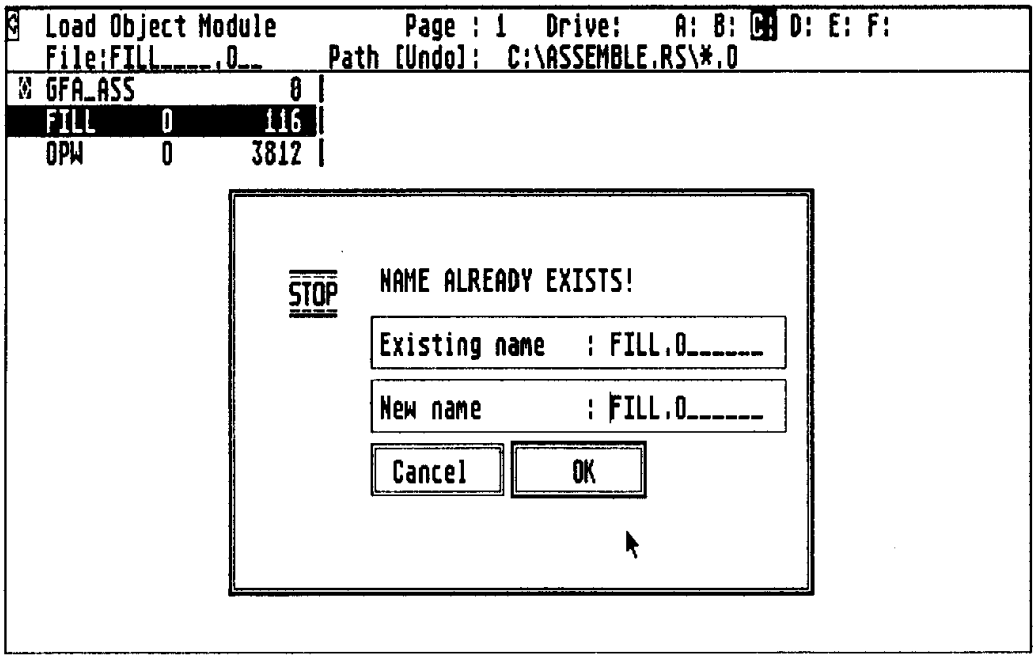


Figure 16 Name Exists Warning

Erase Module

Erase a module. The text is changed to reverse video with the left mouse button. If the <Shift> key is held down, additional names may be selected. The erase routine itself (selecting 'Erase Module' with the left mouse button) is irrevocable!

Save Module

After selecting a module or modules, the selected module(s) will be saved as individual object data files. You must choose the path to the library and the module name.

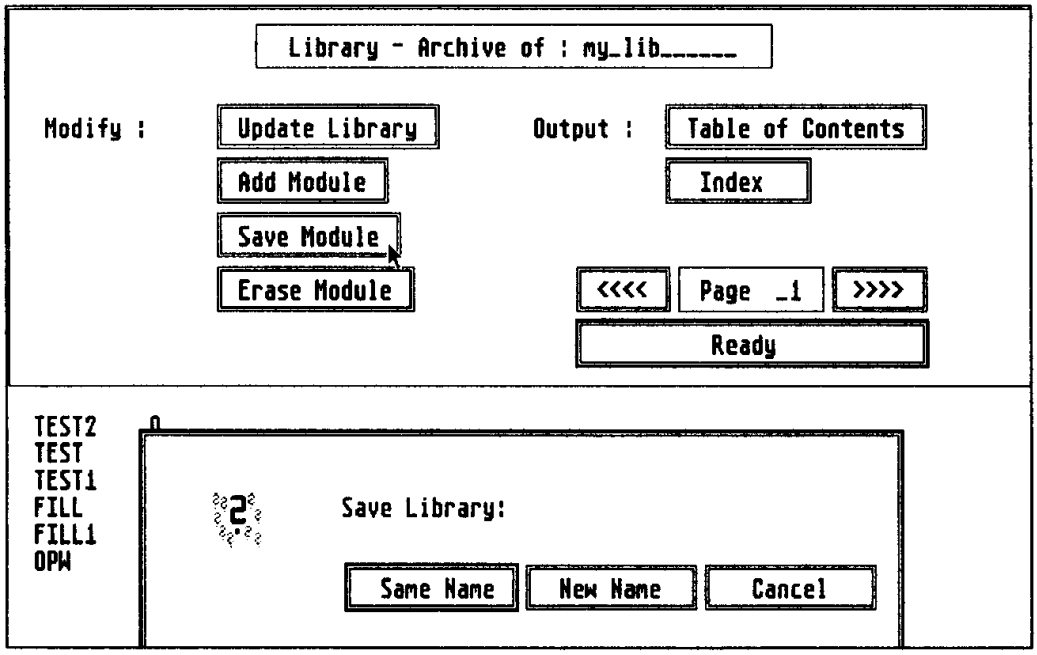


Figure 17 Save Module

Select the modules with the right mouse button.

Creating additional information in a box. (Name, changeable size in bytes date, time).

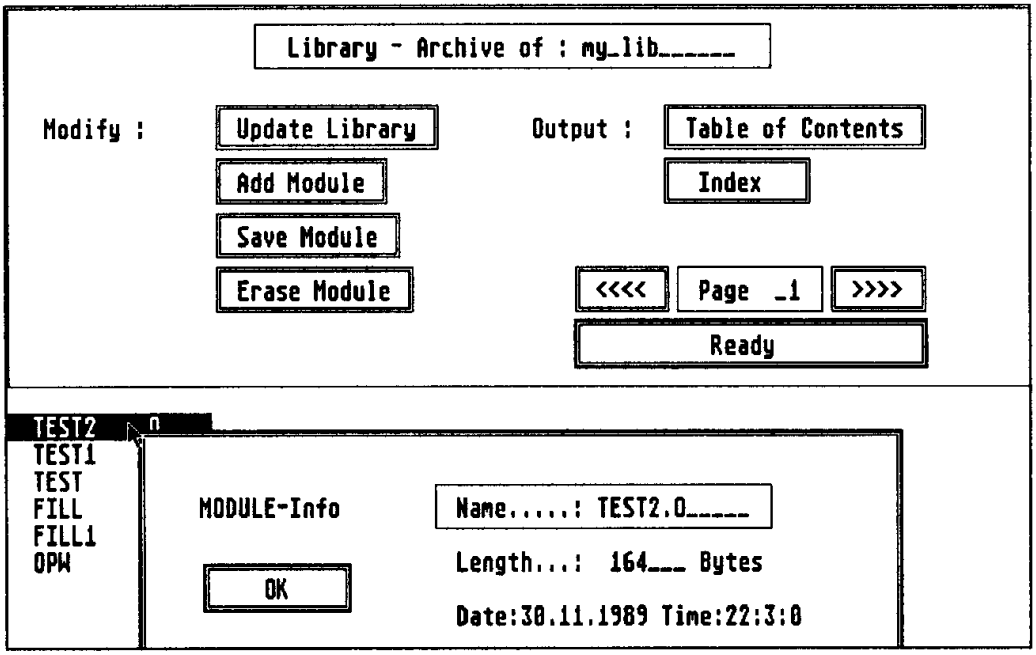


Figure 18 Module Information

Select the Library name with the left mouse button. Library Information Shows number of modules in archive, library size, free memory area (the length of the library may not exceed the length of available memory.).

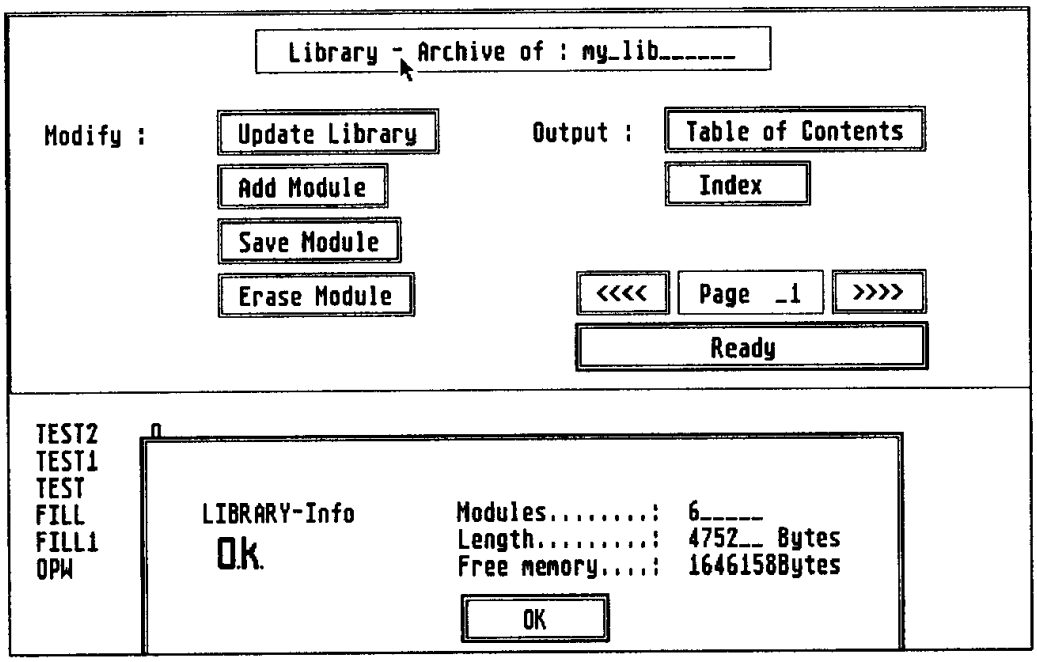


Figure 19 Library Information

Ready

The library is saved to disk in its modified form. A new name may be specified. The user is returned to the main menu of the Editor after saving.

Index

The library access data from the library through an index data file. If no index file exists for a library one must be created. In order to avoid delaying link execution, exit archive management through 'Index' and create the corresponding index file at the same time the modified library is saved.

Warning:

If changes are made to a library and the corresponding index file is not initialized, the linker can become disoriented, leading to errors.

Contents

An ASCII File with a readable form of the index file will be assembled in the same directory as the library. This will enable you to better the internal connections of a library, as shown in the appendix..

Example:

Excerpts from an archive contents directory. (extension: .DIR):

```
GFA Ass V1.2 Index-Data:A:\COPY.\GEMLIB.DIR Date:6.2.1986 Time:1:17:0 S:1
D E P E N D E N C Y T A B L E :
Module.....depends on to:
0 : 83 93 96
1 : 59 60 61 62 63 64 65 66 67 76 77 81 82 83 84 85 91 93 94 96 98 103 105 106 120 121
122
2 : 33 60 61 62 63 64 65 67 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 93 96 104
116 120 121 122
3 : 31 83 93 96 98 103
4 : 31 83 93 96 98 103
...
M O D U L E - S U R V E Y :
0  : channel5.o...0000021 : xmainnw.o....00056a 2 :mallocdb.o...000fce
3  : noascii.o....00137a  4 : nobinary.o...0014ea  5 :nodisk.o.....00165e
6  : nofilesz.o...001866  7 : nofloat.o....00191a  8 :nolong.o.....001b26
9  : nottyin.o....001c46  10 : access.o.....001d52 11 :atoi.o.......001fae
12 : atol.o.......0021be   13 : calloc.o.....0023fe  14 :exec.o.......0025f6
15 : fdopen.o.....00285e 16 : fgets.o......002b0e   17 :fopen.o......002cb2
...
E X P O R T - S U R V E Y :
___atab......105 ___atab......105  ___chini.....83   ___fdecl.....64   ___main.......58
___noasc.....3   ___nobin.....4    ___nodis.....5    ___noflo......7   ___nolon.....8
___open......91  ___prtin.....78   ___prtld......79  ___prtsh.....80   __aflist.....76
__afreeb.....76  __allocc......83  __arith......118  __atof.......109  __blkio......86

G L O B A L and E X T E R N A L Variables from Module :
channel5.o.(Offset: $000002)
        EXPORT: __fds _errno ___cpmrv __errcpm _maxfile __chvec__allocc __freec
                __chinit ___chini __chkc
        IMPORT: __fds _errno _blkfill ___cpmrv __errcpm
 xmainnw.o..(Offset: $00056a)
        EXPORT:___pname      ___atab __fds    ___main _nowildc
        IMPORT: _main _strcpy _sbrk   _exit _brk ___pname___atab___BDOS
                _creata _strcat _opena _lseek _strchr_close__fds
```

Part IV

Additional Information

1.0 Execute Program

A program may be loaded and run from within GFA ASSEMBLER with the function 'Execute Program'. The only condition is that there must be enough free memory for the program to execute properly. If GEMDOS encounters a memory error during loading or exiting the loaded program, the memory error will be displayed and further program loading will be forbidden.

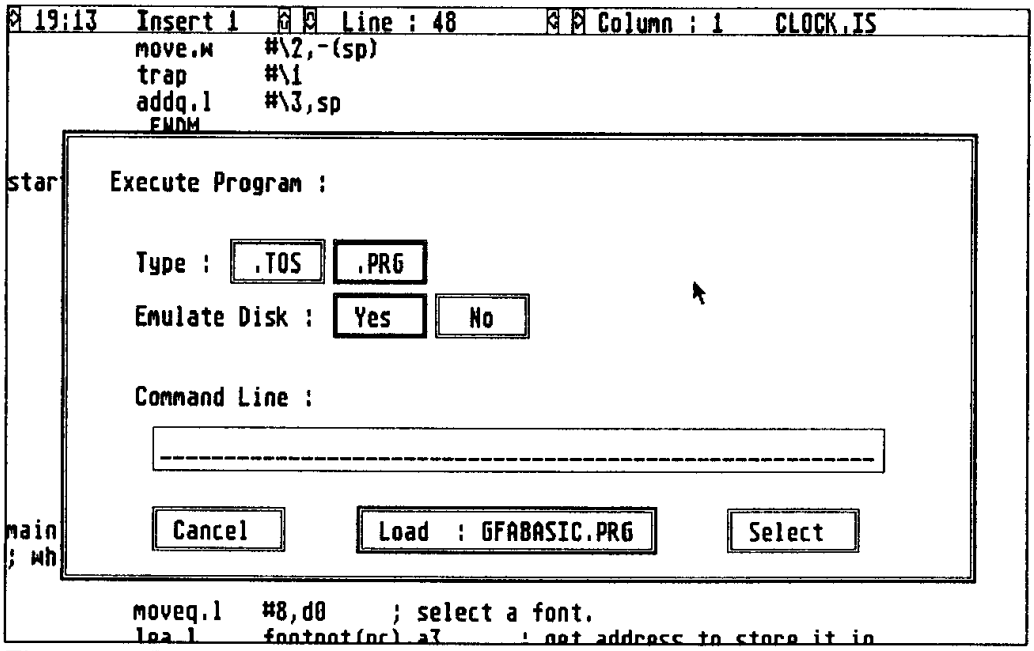


Figure 20 Program Loader Selection Screen.

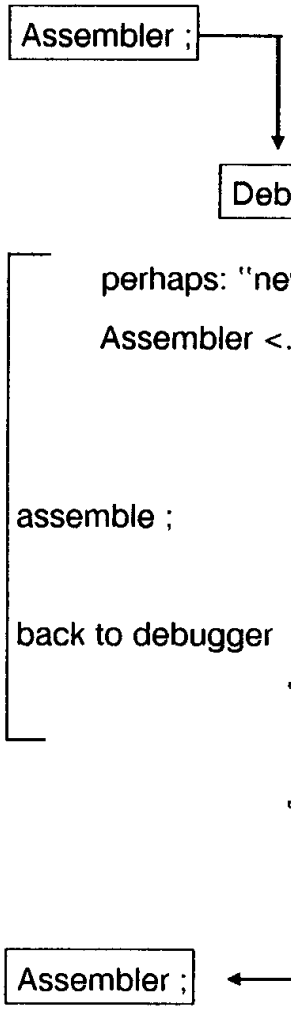
The function 'Execute Program' has four principal uses:

1.1 Loading the Program The program can be tested from within the Editor/Assembler. If an error is encountered (a bus error, address error, etc) the GFA ASSEMBLER will display the appropriate error message and stop loading the program.

1.2 Loading the GFA DEBUGGER

If you need to examine a program in more detail, use the GFA DEBUGGER. The GFA Debugger (GFA-DBG.PRG) can load and run your program. The program to be loaded may be specified in the command line of the Debugger. You may select to run the Debugger from the Editor/Assembler by selecting 'To Debugger' from the Assembler Menu. A special RAM resident of the debugger (GFA-DBGA.PRG) must first be installed from your Auto folder when your ST is booted. This method will permit you to return to the GFA ASSEMBLER when you exit the debugger (with quit).

You will find this very useful when you want to look at the source code of a program or if you want to load a program into the debugger directly from the GFA ASSEMBLER. The flow chart on the next page should clarify this process.



Assembler loads debugger (menu 6,6)

Debugger waits for input

clear debugger's memory

Enter 'asm' command, debugger switches temporarily back to assembler ; you can work in assembler as usual

assembling a program (menu 6,2) or linking (menu 6,4)

back to debugger (menu 1,0)

; to display the program parameters

; you can now analyse the program

; the debugger is left with 'quit'

the memory reserved for the debugger is released again, you have the full amount of RAM available for the assembler.

The framed area may be executed as often as desired. An error in the program can be corrected in the source code, then the program can be loaded into the memory and retested with the Debugger.

1.3 Loading a Compiler

If you use the GFA ASSEMBLER editor to generate source code editor for higher level programming languages, or if you write only time critical portions of your programs in assembly language and other parts in another language, any compiler that uses batch programs (that is one that is executable from another program or a command) may be loaded. The name of the source file to be compiled may be passed to the Compiler by using the command line. In order increase your programming efficiency when using a compiler, be sure that the 'disk emulation' 'Yes' button is activated. The editor of GFA ASSEMBLER will pass the source code found in memory to the compiler as if it were on disk, using the appropriate GEMDOS call. Because the editor does not emulate a single disk drive, but only passes the text in memory instead of reading the disk, the specified file must be found on the disk. More precise information about the emulation mode can be found in Appendix B.

The process of loading a compiler can be better explained with an example: A Pascal Source file called 'TEST.PAS' may be loaded into the text memory of GFA ASSEMBLER. The text may be further edited, and then you may use 'Execute Program' from the Assembler menu. In the command line box enter 'TEST.PAS' and select the 'Yes' button next to 'disk emulation'. Now select 'Select' and select your compiler from the file selection screen. The compiler is loaded and it loads 'TEST.PAS' directly from the GFA ASSEMBLER. Disk access will occur only when the file is to be read or data is saved. If the compiler is interupted by an error, you will be returned to the editor where the error may be corrected. For each subsequent call of the compiler, just select 'Execute Program' <F6> from the Assembler menu and press the <Return> key.

1.4 Loading GFA BASIC 3

GFA BASIC 3 has the capability of switching to the GFA ASSEMBLER. Load GFA BASIC 3 from within the GFA ASSEMBLER using the 'Execute Program' function of the Assembler menu. Place the command 'INLINE' at any place in the GFA BASIC 3 program text. (The syntax for this command is described in the GFA BASIC 3 manual.)

Then position the cursor at the INLINE command and press the HELP key. An additional entry will appear in the menu line, 'ASM ' will appear. Select this item or press the 'A' key. GFA BASIC 3 will pass the address and the length of the area reserved by the INLINE Command to the GFA ASSEMBLER.

When the source code is assembled or linked in the GFA ASSEMBLER, it will automatically be copied into the GFA BASIC 3 program text. This assembly language routine may contain no relocatable addresses. An error usually will generate the appropriate error message. The following points should be noted in the assembler:

If the memory area available is not sufficient for the GFA ASSEMBLER and GFA BASIC 3 with its source code, switching will not be supported. (There should be at least 200K of free memory.) In order to be recognized, GFA BASIC 3 must be loaded from within the GFA ASSEMBLER. Before GFA ASSEMBLER loads GFA BASIC 3, a dialog box will be displayed which can be used to determined which portion of the remaining free memory the GFA ASSEMBLER should maintain for itself. To assembly source code, at least 60 KBytes of memory should remain free. By using this function you can insert machine code programs directly into a GFA BASIC 3 program. These programs must be freely moveable within memory.

In this case when the Assembler is called the button marked 'Load:' will be replaced with 'Load : GFABASIC.PRG'. If the assembler encounters an instruction which is not relocatable or the program becomes longer then the memory area available an appropriate error message will be returned. In order to return to GFA BASIC 3, exit the ASSEMBLER through the 'Quit' function of the File menu. If GFA BASIC 3 is no longer required, exit BASIC by selecting 'Quit' or

'System'. You will be returned to the GFA ASSEMBLER and the loading sequence from GFA BASIC 3 is ended.

2.0 Calling the Memory Resident Debugger

If the debugger is found to be resident in memory (if the resident version (GFA-DBGA.PRG) was loaded from the Auto folder, or if you have previously installed the debugger, a check mark will be placed in front of the 'To Debugger' entry in the Assembler menu. It is then possible to go directly to the Memory Resident Debugger without loading it from the disk. Using the memory resident Debugger is exactly the same as with the Debugger which can be loaded from within the GFA ASSEMBLER.

The function is the same as 'Execute Program' when the Debugger is so loaded into memory. The primary advantage is that the Debugger need not be loaded from the disk and therefore saves time.

Appendices

Appendix A1.

Operation of the Input Fields

The input fields found in the boxes of the GFA ASSEMBLER, can be divided into two groups:

1.) Buttons which are selected with the mouse. Activated buttons are represented as being filled in. There is always a main button which performs a central function. This main button distinguishes itself from the remaining buttons because the frame around it is always thicker. The main button may be selected by pressing the <Return> key. The box may be exited by pressing the <Esc> key.

2.) Fields for input from the keyboard. In these input fields only characters that are meaningful to the corresponding input are permitted. (For example, for an input field which only expects numeric data, only numbers are accepted). A cursor will blink in the input field in which the input is to be made. All characters entered will be inserted to the right of the cursor. The following keys are available for editing the input field:

Cursor Left :	Move the Cursor to the left.
Cursor Right :	Move the Cursor to the right.
Shift + Cursor Left :	Move the Cursor to the beginning of the field.
Shift + Cursor Right	Move the Cursor to the end of the field.
Home :	Move the Cursor to the end of the field.
Clr :	Erase the input field.
Delete :	Erase the character to the right of the cursor. All additional characters are moved one space to the left.
Backspace :	Erase the character to the left of the cursor. All additional characters are moved one space to the left.

If many input fields are found in the box, you can move between the fields with the 'Cursor up' and 'Cursor down' keys.

Appendix A2:

The format of macro tables

A macro table for the editor of the GFA Assembler is ASCII text in which for every reprogrammed event, a sequence of commands is present. Only one command is allowed per line but some commands allow many arguments. A command with its arguments may not be longer that one line.

A sequence of commands for a reprogrammed event starts with the command

PROGRAM event

'event' stands for one of the commands 'KEY', 'KLICK', 'DOUBLE-CLICK' or 'MENU' (see later) with its arguments belonging to them. A key must always be entered after 'KEY' and after 'MENU', one always has to enter the numbers of a menu and of a menu item. After 'KLICK' or 'DOUBLECLICK' no other parameters are expected. This means that the change of the mouse click has nothing to do with the position of the mouse.

In the next few lines are some examples, which define what should happen when the user executes a certain event.

The last command of such a sequence of commands is the command

END

The command 'END' defines the end of a sequence of commands. Between 'END' and the next 'PROGRAM' may lie as much text as you want to put there (comments etc.)

Between the lines PROGRAM...and END you may use the following commands for the change of a command:

KEY key[key[key...]]

The command 'KEY' starts the sending of key events. As arguments you may pass the 'KEY' command as many keys as you want to. 'key' can be one of the following expressions:

n:	A number between 1 and 255 for an ASCII code
$h:	A hex number between $1 and $ff for an ASCII code.
$xxxx0000:	A hex longword. In its upper word, in the lower byte is the scan code and in its upper byte are the special keyslike Control/Shift.
'string':	Characters consisting of one or a few ASCII characters.
UP,DOWN,LEFT, RIGHT:	Cursor movements
CLO1,COLE:	To the beginning/end of the line
LN1,LNE:	To the beginning/end of the text
HOME,END:	To the beginning/end of the page
PGUP,PGDN:	Page up/page down
WORDL,WORDR:	Word left,Word right
DEL,SDEL:	Delete,Shift + Delete
BSP,SBSP:	Backspace, Shift + Backspace
WORDDEL:	Erase word
LNRDEL,LNBDEL:	Erase the rest/start of the line

RETURN:	Return
INSLN:	Insert a line
TAB:	Tab
F1..F10:	Function keys
UNDO:	Undo
HELP:	Help key
PRTSC:	Shift + Help
TXT+,TXT-:	Change text, forwards/backwards
LTXT,RTXT:	Change to the left/right text
TXTW+,TXTW-:	Change the size of the text window bigger/ smaller
SETM1..SETM9:	Set marks
GETM1..GETM9:	Jump to marks

KLICK [x,y]

The command KLICK executes a mouse click at the pixel coordinates x,y. x and y can be left out and a mouse click will be executed at the current position.

DOUBLECLICK [x,y]

The command DOUBLECLICK executes a double click at the pixel coordinates x,y. (x and y may be omitted).

MENU menu menu item

The command MENU executes the menu menu,menu item.

LINE N or LINE "label"

The Command LINE sets the cursor to the line 'n' or in the line in which it found 'label' as a label or as a macro.

LABEL "label"

The command LABEL allows the search of a symbol definition from the position of the cursor onwards.

COLUMN n

The command COLUMN puts the cursor in the column 'n'.

TEXT n or TEXT "name"

The command TEXT shows the text with the number 'n' or the text with the name 'name'.

Appendix A3.

The Standard Printer Settings

If the GFA ASSEMBLER finds no data file with the name 'PRINTER.CFG' when it is loaded, and if no '.CFG' data is loaded from within the assembler, the GFA ASSEMBLER will print using the following default settings:

$0D,$0A	Carriage return
$0C	Form Feed
$1B,'-',$01	Turn on underline
$1B,'-',$00	Turn off underline
$07	End printing

All characters are output to the printer exactly as they appear in the text.

Appendix A4.

The Printer Configurations Table.

The printer configuration tables are in ASCII format and use the filename extension '.HEX' before being reconfigured into a '.CFG' data file through the Configuration Table 'Compile' button. This text file is comprised of four areas:

Printer name
Printer type
Control sequence for formatting
Character conversion table

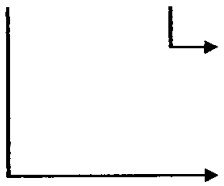
The areas have the following contents:

Printer name: One line with the name of the printer. This area is not used in the present version but must be included for compatibility.

Printer type: 6 Hex Bytes that characterize the printer type. This area is not used in the present version but must be included for compatibility.

Control sequence for formatting: This is a table with the following construction:

- 01	xx,xx,...	Control character that will be exchanged for carriage return.
- 1a	xx,xx,...	Control character that will be exchanged for 'underline on'.
- 1b	xx,xx,...	Control character that will be exchanged for 'underline off'.
- 1e	xx,xx,...	Control character that will be exchanged for form feed.
- 21	xx,xx,...	Control character that will be exchanged.



Any number of Hex Bytes.

Number of the function. These entries must be in ascending order. Other function numbers are also translated, but are not used in this version of GFA ASSEMBLER.

Character conversion table: This is a table in which all characters are listed one after another for which one or many characters are to be output. Every entry is made up of the Hex Code of the character and the order in which the character is to be output.

As an example of such a text may be found in the file 'PRINTER.HEX' on the program disk. This table can be modified for other printers.

Appendix A5: When Memory Fails

The GFA ASSEMBLER uses the available memory to execute functions quickly on the ATARI ST. This means that if the memory is nearly full, many functions will execute slower. In this case, the functions 'Create File Table', 'Compile' and the automatic screen save will eventually be stopped. (See also page 173)

Appendix B.

Data Format for Assembler and Linker

Appendix B1:

Format of Executable Programs

An executable program, one that can be run directly from the desktop by clicking on it with the mouse cursor, is comprised of:

Program Header	(28 Bytes)
Text Segment	(Program Code)
Data Segment	(Program data)
BSS Segment	(not contained in the data, is first loaded when the operating system is set up)
Symbol table	(optional, either contains all symbols used during assembly or only global symbols, helpful during debugging)
Relocatable Info	(Depending on memory usage, a program is loaded into the operating system at various addresses. The programmer can not know the programs operational position during execution. Therefore, all programs are so written as though they start at memory address 0. The operating system calculates the relocation information from the absolute address into the physical address before the program is executed.)

Header Format: **lfd.Program**

Position:	**Contents:**
$00 = 0	$601a 'Magic'
	ID code for GEMDOS
$02 = 2	Length of the Text segments
$06 = 6	Length of the Data segments
$0a = 10	Length of the BSS Segments
$0e = 14	Length of the Symbol table
$12 = 18	10 Null bytes
$1c = 28	Beginning of program

Relocation Information

The relocation information begins at an even address with a long word which contains the relative distance to the first absolute address of the program from the beginning of the program. The coordinates to all additional absolute addresses are distances in bytes. If the distance between 2 absolute addresses is larger than 254 Bytes, then 'n' times 1 is set.

The following applies: distance =n * 254 + rest

Where 'n' is a number representing the distance divided by 254 and 'rest' is a number that is smaller than 254. Therefore n+1 bytes are required.

Appendix B2:

Object Module Format (DRI Format)

Object modules can not be directly executed, they must first be linked into an executable program with a linker, possibly, in combination with other modules. They do closely resemble the construction of a program:

Program Header	(The same as with an executable Program)
Text Segment	(Absolute address relative to the Segment Base)
Data Segment	
BSS Segment	
Symbol Table	(Not identical to a PRG Symbol table)
Relocations	(No connection with program relocation Information)

A linker gathers the text segments of the module into a combined text segment with the data segments and BSS segments. When the object module is created, its length is not known. All absolute addresses are referenced to the SEGMENT start and are first calculated in reference to the PROGRAM start by the linker. This applies to the symbol values in the symbol table as well.

The relocation information of a program in relation to the relative distances from the description of the object module is unsuitable. The relocation information and data segment of the module are all in all made up of words and long words with the following meanings:

$0000	Contains no information
$0007	Contains no information. Marks the beginning of an opcode (all opcodes are words at whichthe operand is inserted.) The GFA ASSEMBLER/Linker can work withthis information, it will not produce anything in its object module.

$0005 Shows the relative offset (a long word) by which the corresponding relative address in text or data segments must be relocated. An additional word will follow with the following possible contents:

$0002 Absolute Address for Text Segment
$0001 Absolute Address for Data Segment
$0003 Absolute Address for BSS Segment
$xxx4 Unresolved reference. The name of the appropriate symbols is reachable as follows:

($xxx4 - 4)/8 = Number of the name of the module with the proper symbol table (begins with 0).

The address relative to the symbol table beginning is:
Address = (($xxx4 - 4)/8)*14

$yyy6 An unresolved reference or a data segment occurs at a corresponding position in the text segment. Care should be used that if a PC relative word address is used to move beyond the segment limit, or when working with an as yet unknown address, because the range of this type of addressing (+ or - 32 KByte) can be over extended. The linker will warn the user if necessary.

The appropriate Symbol address is calculated with the unresolved long word addressing:

rel.Symbol address = (($yyy6 - 6)/8)*14

Construction of the Symbol Table

Each entry in the Symbol table is $e (=14) Bytes long and constructed as follows:

Address to entry start:		Contents	Bit:meaning:(if Bit=1)	
$0...$7	= 0...7	Symbol Name (filled with 0)	0	Symbol of the BSS Segments
			1	Text Segments
$8	= 8	Symbol Status	2	Data Segment
$9	= 9	not used	3	Symbol not defined
$a $d	=10...13	Symbol Value	4	Register
			5	Global Symbol
			6	Constant Symbol
			7	Symbol is defined

Notes:

The Symbol Type COMMON Not supported by the GFA Linker.

Warning: Symbols are basically only 8 characters long during linking. If longer symbols are encountered during assembly they will be shortened to 8 characters without any warning.

Appendix B3: ABS File Format

ABS files are created by using the 'A' Option of the linker. They have the same format as an executable program with the limitations described in the linker documentation. Besides this the header that was described is to be paid attention.

Appendix B4: IMG File Format

Occasionally it is not possible or desirable to relocate programs before they are executed. If the user creates programs that are to be freely relocatable in memory, no absolute addresses (except of course hardware addresses) may be used. Header and relocation information is superfluous. Such files can be created with the IMG Option of the linker. The linker checks if absolute, or relocatable addresses are contained within the program and warns if one is found.

Header and relocation information is never created, a symbol table can be inserted as desired.

Hint: Commands passed to GFA-BASIC 3 with the INLINE Command also have this format.

Appendix B5: Library Construction

Libraries used by the linker contain object modules and have a very simple structure: They begin with an identifications word ('Magic', $ff65), followed by the sequential object module in the format described in Appendix B1. Each object has an additional header which is 28 Bytes in length and made up as follows:

Address Header Beginning		Contents
$00...$0d	= 0...13	Module Name
$0e...$0f	= 14...15	Date of creation in GEMDOS Format
$10...$11	= 16...17	Time of creation in GEMDOS Format
$12...$13	= 18...19	Null bytes
$14...$15	= 20...21	$01b6
$16...$19	= 22...25	Length of object module
$1a...$1b	= 26...27	Null bytes
$1c...$37	= 28...55	Object module Header
$38...	= 56...	Object module

The end of the library is marked by a null word.

Appendix B6:

Index Files (for Libraries)

The linker understands the construction and all the exportable symbols when the index file of a library is used. The index file can be converted into an ASCII file containing the same information by using the Management menu of the Assembler/Editor. These two files use the extensions '.NDX' and '.DIR' so they are not confused.

Construction of these files:

$ff75	Identification ('Magic')
$xxxx	Number of symbols exported from the library (word)
$yyyy	Number of modules contained in the Library (word)

Module table	1)
Export table	2)
Dependency table	3)

1) One module entry is 20 Bytes long and contains the offset of the module (the last four Bytes) relative to the library beginning next to the module name (16 Bytes). The names of the modules in the library are sorted alphabetically.

2) One Export entry contains the export name (8 Bytes) and the number of the module to be exported as a long word (4 Bytes).

3) When a module is combined this will contain mainly unused references that concern exporting other modules. A mask concerning every module exists which contains information about which modules must be included in combination with the module selected. Each module corresponds to one Bit of this mask.

Example: One index file to a library with 19 modules is to be created:
Each mask is 3 Bytes long (corresponds to 24 bits of which only 19 are necessary).

Each mask has the following construction:

0000 0000 0011 1111 111	Module	x,y,z=0:
0123 4567 8901 2345 678.	Number	Module not necessary

xxxx xxxx yyyy yyyy zzzz zzzzaccompanies x,y,z = 1:

Byte1	Byte 2	Byte 3	Mask bit	Module is necessary

There are 19 such masks present. The format of the ASCII format index file is described in the Archive management documentation.

Appendix B7:

Tokenized Source Code Format

A special feature of the GFA ASSEMBLER is it's tokenized Source Code format. Knowledge of this format is not usually necessary for the user. Those developers that may be interested in this format can obtain detailed information about it from GFA.

Appendix C.

Error Messages from the Assembler and Linker

As already explained in the operating procedures to the Assembler/ Linker, not every inconsistency during assembly will lead to a work stoppage. Therefore, small errors will be taken in stride. They are always made known to the user.

The following is a list of all error messages with the possible consequences. For more specific information, refer to the appropriate Chapter.

Error message:		Explanation:
Syntax error	1)	A command, Pseudo-Opcode etc, was entered incorrectly.
Illegal address type	1)	An address type not in use is expected with an Operand for a processor command.
Illegal Extension	1)	Incorrect command value (.b,.w, .l), not every command works with all types of data width's.
Command not complete	1)	Operand or dgl. is missing.
Illegal value	1)	In a value area-overwriting, illegal data type, syntactically incorrect calculation etc.
Not interpretable Pseudo Opcode	1)	Input of unknown PseudoOpcode
Symbol Name Too long	1)	Symbol name (those with "\~" also) may only be 16 characters long at max.
Branch out of reach	*)	PC-relative branch out of reach.
Symbol redefined	*)	-----

Undefined symbol	*)	If an executable program is created with the assembler all symbols that are used must be defined.
Filename not specified	*)	Filename (with .Include, linking) is missing.
Relocation attempt	*)	In IMG mode of the linker and during program passing to GFA BASIC absolute addressing is not permitted.
Symbol necessary	1)	Symbol (as function–operand) was not given.
Incorrect number of parameters	*)	---
Include not possible	*)	Include attempt on a non existent source text file.
Memory full	*)	See below
Division by 0 .EVEN necessary	*)	Should words (processor command data) be written at odd address .EVEN must first be executed.
Only one assembly possible	*)	In the linker control field the "^" command is called more than once.
Pass differentiation	*)	Consult .IF–documentation.
Branch to next command	*)	Only in the branch optimization mode! Is not optimized, Branch–commands with word offsets are coded.
.ENDIF without .IF	*)	---
.IF without .ENDIF	*)	---
.ELSE without .IF	*)	---
LOOP not closed	*)	.ENDR was forgotten
Macro redefined	*)	---
Macro not defined	*)	Macro's must be defined before they are called.
.ENDM without .MACRO	*)	---
overflowing .ENDM/EXITM	*)	---
Unknown segment type	*)	Consult SECTION command

Error during linking	*)	Fatal linking error (diverse reasons)
Check segment type	*)	.DS.x command in text or data segment or .DC.x or 68k command in BSS or ABS segment.
Incorrect Segment Definition	*)	Redefinition of a segment or definition of a segment with NR.>63
Error during protocol	*)	File access error during assembler protocol creation.
Relative addressing out of reach.	2)	Linker can not detect 16 bit references because of too large an address (> approx. +/- 32kByte)
ASSERT/FAIL	*)	Assembly stopped by user command
Too many path's specified.	*)	There are a maximum of 16 Include-Search_path's allowed (search in the order of input).
Incorrect page format	*)	Meaningless page format or page buffer too small (Choose a smaller format or in the printer menu increase the page buffer!)

Symbol meanings:

1) Is also reported during input
2) Exclusively linker error message
*) fatal error (Work is interupted.)

Procedure with full memory:

The GFA Assembler is memory–orientated in all respects. It can therefore happen, especially with large amounts of data and the smaller computer models, that the amount of available memory is insufficient. We therefore suggest a few emergency measures to save some memory space in such an event. The following should be tried in this sequence when your RAM space is rapidly decreasing:

– Disable all desk accessories, RAM disks, operating system patches and other "memory gobblers"! (if necessary with a system reset!)

– Do not use ASM from the debugger or BASIC!

– Remove all surplus texts from memory!

– Reduce the size of the spooler and page buffers in the printer menu!

– Use INCLUDE from disk or hard disk! (For source files read in during assembly/linking, 7 bytes per line are required by the editor's memory management)

– Avoid assembly during linking! (Generate source modules separately first, then link!)

– Split your program into object modules for the linker!

– Remove (using "Search and replace") the comments from your source code!

– Buy a RAM expansion!

Appendix D

Cross Assembly

It is possible to generate programs and object modules for the Commodore Amiga computer by inserting the pseudo opcode 'AMIGA' into an assembler source file. On encountering 'AMIGA', the assembler quits normal assembly and enters into cross assembler mode. It is therefore advisable to place 'AMIGA' at the beginning of a source text.

The different program format of AmigaDOS explains the following peculiarities of segmentation in cross assembler mode:

Every SECTION used produces a separate program segment at object code level. It is neither necessary nor desirable to group sections into text, data or BSS segments.

The SECTION instruction can take the name of the segment as an additional parameter with text, data and BSS segments. AmigaDOS linkers can combine segments with identical names.

Syntax: SECTION, Nr,Type,'SegmentName'

An entire object module can be assigned a module name (for libraries!) in addition to its filename by using IDNT.

Syntax: IDNT 'ModuleName'

An AmigaDOS program is loaded segment by segment. The relative position of the segments is not predefined and depends on the current state of the system. PC-relative addresses between segments are therefore illegal. The assembler follows this rule in the optimisation modes and reports any mistakes made by the user.

Developing AmigaDOS programs requires a set of INCLUDE files with all system variables predefined. They are not included in the GFA ASSEMBLER ST for copyright reasons. You can, however, copy the original Includes to ST disks, load them as ASCII files and convert them – with minor syntax modifications – to assembler source text mode. Another option is to use the includes that come with the GFA

ASSEMBLER AMIGA. These are derived from the original includes and are supplied as '.IS' files. The formats are 100% compatible. It is necessary, however, to modify the directory paths used in the includes (different device names, '/' instead of '\', long filenames!).

To transfer programs to the destination computer, you can use the appropriate utilities for reading MSDOS disks on an AMIGA (Dos-2-Dos, for example). An easier way is to use parallel transfer via the printer port. A driver (with a transfer rate of about 50kb/second!) is included with the GFA Assembler. The necessary cable, together with receiving software for the Amiga, can be obtained from GFA Data Media. Please phone or write for further details.

WARNING: This is a special cable. Do not attempt to transfer data with a home-made or a centronics cable, as your Atari ST (and possibly your Amiga, too) would be destroyed! We expressly reject any responsibility for damage resulting from a wrong use of the transfer facility!

To use the cable, the following instructions must be employed:

OUT UNLINK

When the assembler encounters OUT or UNLINK in the source text, the saving of the generated file is suppressed and instead it is transferred via the parallel port. UNLINK aborts the connection to the Amiga after the transfer, OUT continues it so that several files can be assembled and transferred in a row. To generate a number of programs or object modules automatically, it is a good idea to use the control file to control the linker:

Example:

=modul_1.is =modul_2.is

Executed as a linker control file, this generates the relevant code before quitting with an error (because of the wrong format for the ST or the missing files). This, however, has no consequences for the correct automatic generation of the required files.

Once the required code has been generated and transferred, it can be further processed on the Amiga:

* execution of executable programs, or * linking in standard AmigaDOS format

For debugging on the Amiga we recommend the GFA DEBUGGER AMIGA which is included in the GFA ASSEMBLER AMIGA development pack. (Symbolic debugging!)

Assembler Index