# X16 Edit implementation notes

December 6, 2020

## 1 Introduction

This document contains some general notes on the implementation of X16 Edit. For further details, see the source code comments.

## 2 Variables

### 2.1 Zero page

The zero page is primarily used to store global variables. The first zero page address used by the program is $22.

The following variables are of special interest in understanding the program:

Table 1: Important zero page variables

| **Name** | **Description** |
|---|---|
| CRS_BNK | RAM bank part of pointer to text buffer, cursor position |
| CRS_ADR | AddressL and AddressH of the above-mentioned pointer |
| CRS_IDX | Index (offset) within memory page |
| LNV_BNK | RAM bank part of pointer to text buffer, the first visible character of the current line |
| LNV_ADR | AddressL and AddressH of the above-mentioned pointer |
| LNV_IDX | Index (offset) within memory page |
| SCR_BNK | RAM bank part of pointer to text buffer, the first visible character of the screen |
| SCR_ADR | AddressL and AddressH of the above-mentioned pointer |
| SCR_IDX | Index (offset) within memory page |

The CRS_XXX variables are, naturally, used to keep track of the position of the cursor in the text buffer.

On screen refresh, the program starts to print from SCR_XXX. On line refresh, output starts from LNV_XXX.

SCR_XXX and LNV_XXX are also used for scrolling the text vertically or the current line horizontally. By moving SCR_XXX to the start of next line, all screen content is, for example, moved up one row.

### 2.2 Local variables

Memory page $0400 is used for local variables. The program is designed to run in ROM, and no variables can be therefore be embedded in the code segment.

### 2.3 I/O buffers

The program uses three buffers.

The input prompt buffer is stored in page $0500. The input prompt is used when the program needs input from the user.

The file name buffer is stored in page $0600. The file name buffer holds the current file name.

The file status buffer is stored in memory addresses $0700–$077F. It holds the result of the last disk status read.

The clipboard buffer is stored in banked RAM, bank number 1, start address $A400. The buffer is 3 kB.

## 3 The text buffer

### 3.1 Memory blocks

The text buffer is conceptually divided into blocks of 256 bytes each, what you normally call a page (as in "zero page"). The least significant byte of the address where a block starts is always 0.

The blocks are organized as doubly linked lists, holding a reference both to the previous and the next block. Each block also has the length of the string stored within it.

The memory model makes the operation of the program significantly more complex, but it is necessary to keep up performance when editing large text files—one of the main design goals of the program. If the text buffer were treated as a continous string of bytes, inserting or deleting charactes at the top of the file would require the program to move all subsequent data either back or forth. This is fine if the the text buffer is short, but would lead to significant performance drop as the buffer grows in size.

Table 2: Memory block content

| **Offset** | **Description** |
|---|---|
| $00 | Previous block (bank number) |
| $01 | Previous block (addressH), 0 = BOF |
| $02 | Next block (bank number) |
| $03 | Next block (addressH), 0 = EOF |

| | |
|---|---|
| $04 | Text length |
| $05–$FF | Text data |

When inserting a character into the text buffer, the program first checks the length of the current block. If there is room for one more character, it is inserted there.

If the block is full, but there is room in the next block, the program will make room for the new character by moving text to the next block.

If the next block is also full, the program allocates a new block of memory that is linked in after the current block. Characters are moved from the current block to the newly allocated block to make room for the new character.

When deleting a character, the length of the current block is decreased by one, and the characters after the deleted characters are moved one step left in the block.

In this way the program never needs to copy or move large amounts of data even if the text buffer is filling up almost all of banked RAM.

## 3.2 Text buffer defragmentation

One downside of the text buffer strategy described above is that the text buffer might get fragmentated as you move around inserting and deleting text. By fragmentation I mean memory blocks, other than the last block, not fully used.

To counter this issue, the program has a defragmentation routine. That routine is run once every interrupt cycle if there is no keyboard input to handle. For each such invocation the routine defragmentates one block. Defragmentation starts from the head of the text buffer and continously works through all allocated memory blocks.

The defragmentation routine first looks at the length of the text in the memory block it is working on. If the length is less than 251 bytes, it moves data from the next block until full. If the next block is completely empty after this, it is deallocated and marked as free.

## 3.3 The memory map

The program uses a memory map to keep record of the status of each memory block (free or allocated). The map resides in banked RAM, bank number 1, address $A000.

In the map each memory block is represented by one bit, making it 1 kB in size (256 banks · 32 blocks/bank = 1,024 blocks).

On startup the program allocates the head of the text buffer (bank 2, block start $A000). During the operation of the program blocks may be allocated or freed with the functions mem_alloc and mem_free. The head of the buffer is never deallocated.