MicroSoft Word file format

second, revised edition

M.J. Carels,A.G. StarreveldDepartment of Computer Science University of Amsterdam

e-mail: {decvax, philabs, seismo}!mcvax!uva!{maarten,dolf}s-mail: Kruislaan 409, 1098 SJ Amsterdam, The Netherlands

1. Introduction.

This document describes the structure of the files produced by MicroSoft WORD (versions 1.00 and 1.05). These files are of type 'WDBN'. This knowledge has been gathered by looking at such files, and trying to interpret the bytes in the file. All information is believed to be correct, but no responsibility for errors or omissions can be taken. Please let us know if you find errors in this document, or if you find more about the structure of the file. The file format used by MS Word for other computers than Macs may, or may not be the same.

We will describe all structures of the file format by means of C structure declarations, as this is a convenient way to describe things. In the structures some basic types appear. These are:

`ubyte`	8 bits unsigned number
`byte`	8 bits signed number
`ushort`	16 bits unsigned number
`short`	16 bits signed number
`ulong`	32 bits unsigned number
`long`	32 bits signed number

No alignment is to be assumed, the only bytes present are the ones described. In all structures byte addresses (decimal) are included as comments.

2. General file structure.

A MS Word file of type 'WDBN' can be divided into seven different main parts. We will call these parts the "**header**", "**text**", "**character formats**", "**paragraph formats**", "**division blocks**", "**division list**" and "**page list**" respectively. These parts appear in the above mentioned order in the file's data fork. The resource fork is always empty, i.e. not allocated.

The file can be seen as being built from basic blocks, each 128 bytes long. This is the case for all parts of the file, although it does not appear to be significant for the text part. This implies that the size of an MS Word file is always a multiple of 128 bytes, though within each of the parts mentioned, the last bytes in the last block belonging to a certain part may (and usually will) contain garbage.

In several sections sizes, dimensions and distances are present. In all such fields these dimensions are given in 'basic units'. A MS Word basic unit is 1/20 of a point, or 1/1440 of an inch (a point equals 1/72 of an inch).

3. Header.

The header part consists of a single block of 128 bytes. It contains pointers to most parts of the file. The header can be defined in terms of a C structure as follows:

```
   /* *     This structure starts each MS-WORD 'WDBN' file. */struct     _header
            {/* 00 */                 ushort h_1;  /* always 0xfe32 *//* 02 */
            ushort  h_2;              /* always 0 *//* 04 */    ushort     h_3;
            /* always 0xab00 *//* 06 */     short h_unk1[4];  /* always 0 *//* 14
*/          ulong   h_ET;             /* Position of byte past text *//* 18 */
            ushort  h_par;            /* first paragraph block # *//* 20 */ushort
            h_div;  /* first division info block # *//* 22 */ ushort     h_div1;
            /* same *//* 24 */        ushort h_divlist;  /* first division list
block # *//* 26 */     ushort         h_pagelist;  /* first page list block # *//*
28 */       ushort  h_unalloc;        /* first unallocated block # *//* 30 */
            short   h_unk2[17];       /* always 0 *//* 64 */    ulong h_tlength;
            /* Length of text *//* 68 */    ulong h_tlength1; /* same *//* 72 */
            short   h_unk3[28];       /* always 0 */};typedef struct _header
MS_Head;
```

The `h_ET` field contains the address within the file of the byte just past the last character in the document text, i.e. the “text” part. The `h_tlength` field contains the length of the “text” part in bytes. The `h_par` field gives the block number (remember a MS Word block is 128 bytes long) of the first block that contains paragraph formats. Every division has its own block containing margins, page number and that kind of stuff. The `h_div` field contains the block number for the first division block. For some reason it is stored twice. The connection between the text and the division blocks is made through the division list. The `h_divlist` field gives the block number for the first block in the division list. The last block in the file contains the page list. In the page list the position of the first character in the first line of each page is stored. It is this list which is updated when you issue a “Repaginate” (COMMAND-J) command. The small ‘=’ signs in the margins come also from this list. The `h_pagelist` field gives the first block for this list. The block number of the first “unallocated” block is stored in the `h_unalloc` field. The file is also `h_unalloc` blocks long.

4. Text part.

The text part contains a complete representation of the text in the document, including running heads, footnotes and pictures. The text is represented in the order in which it occurs in the document, in the extended Macintosh ascii character set. Some ascii values have special meaning however:

0x01 page number ((page) glossary)0x05 auto numbered footnote reference ((footnote)
glossary)0x0b Forced new line within paragraph0x0c End of division or forced new
page0x0d End of paragraph0x1f Optional hyphen

The above implies that to extract a text only version of an MS Word file one only needs to extract the text part of the file, possibly replacing some of the special characters with others, depending on what you want. If you do nothing, you will certainly get very long lines, since you will get a newline character only at the end of each paragraph, so perhaps you want to do some line folding. Pictures are stored within the text part, along with a header. The picture is a single paragraph by itself. The paragraph format run pointing to the picture has a bit set to indicate the corresponding paragraph is a picture.

5. Format runs.

Everything related to the layout of the text is stored in what we will call "format runs" and "format descriptors". A format run consists of several bytes of formatting information, described below (section 6 and 7). A format descriptor consists of 6 bytes. It is described by the following structure:

```
    /* *      This structure defines a format descriptor. */struct    _fdescriptor
              {/* 00 */                ulong   fd_start;   /* start of text for next
run *//* 04 */  short   fd_run;          /* pointer to this format run */};
```

Each format block starts with the offset in the text part where the formats of this block start. After the initial start a number of format descriptors follow. The rest of the format block contains format runs. Both the format run and the format descriptor must be contained in the same block. A new block is allocated if either one does not fit.

The last byte in a format block (offset `0x7f`) contains the number of format descriptors present in the block.

The format runs are stored preceded by a byte count. This bytecount gives the number of bytes in the format run that are actually stored in the file. The other (not stored) bytes of the format run contain the default value. File size is reduced by not storing seldomly used fields.

The `fd_start` field is a pointer in the text part of the document. The **next** format applies from there. The `fd_run` field is an offset (relative to byte 4 in the format block) to the format run.

6. Character formats.

The character format runs define how the characters in the text look. This includes properties like the font, size and style of the characters. The character format runs are 6 bytes long, although not all 6 need be stored. One field needs special attention. The font number is split in (at least) two pieces. The low order 6 bits are in the `cf_font` field. This fits most standard fonts, as they have small numbers. More exotic fonts have larger numbers. The extra bits are stored in the high order 3 bits of the `cf_flags2` field. The meaning of the bytes in the format run is:

```
      /* *      This structure defines character formats. */struct      _cformat
                {/* 00 */ ubyte          cf_unknown;  /* seems always 0x80 or 0x00
*//* 01 */      ubyte    cf_font;        /* font number, some flags *//* 02 */ ubyte
                cf_pointsize;            /* times 2, 0 = default *//* 03 */   ubyte
                cf_flags1;               /* more flags *//* 04 /* ubyte cf_flags2;
                /* more flags, more font # *//* 05 */ byte  cf_position;      /* > 0
super, < 0 sub script */};typedef struct _cformat MS_CFmt;

      /* macro for extracting the font number */#define CHF_FONT(x) (((x)->cf_font&0x3f
(((x)->cf_flags2&0xe0) << 1))/* values for the cf_font field */#define  CHF_BOLD    0x8
                /* bold bit */#define    CHF_ITAL      0x40  /* italic bit *//* values f
the flags1 and flags2 fields */#define   F1_UL  0x80  /* underlined */#define F1_SC 0x0
                /* Small Caps */#define  F2_OL   0x10  /* outline */#define    F2_SH 0x0
                /* shadow */
```

The first field in a character formats format run seems to take only the values 0x00 and 0x80. The meaning of this field is unknown.

7. Paragraph formats.

The fourth part of the file contains the paragraph formats. The format runs start with normal paragraph formatting information. Thereafter follow the “tab definitions”. As many tab definitions as needed will be in the format run. The tab definition is described by the structure below:

```
      /* *      This structure defines tabs */struct  _tformat {/* 00 */      ushort
                t_position;              /* tab position *//* 02 */     ushort
                t_flags;  /* type of tab stop */};typedef struct _tformat MS_Tab;

      /* values for the t_flags field */#define T_ALIGNMSK  0x6000      /* mask for
tab alignment */                         /* alignment values: */#define  T_LEFT
                0x0000   /* left aligning tab */#define     T_CENTER    0x2000
                /* center aligning tab */#define      T_RIGHT     0x4000      /*
right aligning tab */#define             T_DECIMAL    0x6000      /* decimal tab
*/#define       T_LEADMSK 0x0c00         /* mask for tab leader */            /*
leader values: */#define  T_BLANK        0x0000  /* blank leader */#define     T_DOTS
                0x0400   /* dotted leader */#define   T_DASH      0x0800      /*
dashed leader */#define   T_LINE         0x0c00  /* line leader */
```

The format run is defined by the structure below:

```
      /* *      This structure defines paragraph formats. */struct      _pformat
                {/* 00 */ ushort         p_flags;      /* some flags *//* 02 */
                ushort   p_unk1;         /* always 0 *//* 04 */   ushort
                p_right;  /* right indent *//* 06 */  ushort      p_left;      /* left
indent *//* 08 */         ushort         p_first;      /* first indent *//* 10 */
                ushort   p_line_spacing;/* line spacing (0 = auto) *//* 12 */ ushort
                p_before; /* space before *//* 14 */  ushort      p_after;     /*
space after *//* 16 */    ushort         p_rhead_pict; /* running head & picture info
*//* 18 */      ushort   p_unk2;         /* always 0 *//* 20 */   ushort      p_unk3;
                /* always 0 *//* 22 */   MS_Tab p_tabs [0]; /* list of tab
descriptors */};typedef   struct _pformatMS_Fmt;

      /* values for the p_flags field */#define PF_FOOTMASK 0x7f00      /* mask for
footnote info */                         /* values unknown */#define     PF_JUSTMASK
                0x00c0   /* mask for justification */            /* justification
values: */#define        PF_LEFT        0x0000 /* left justified paragraph#define
                PF_CENTER 0x0040         /* centered paragraph */#define PF_RIGHT
                0x0080   /* right justified paragraph */#define  PF_JUST     0x00c0
                /* justified paragraph */#define      PF_KEEP     0x0010      /* keep
with next paragraph */#define            PF_KEEPL     0x0020      /* keep lines
together *//* values for the running_head field */#define   RH_MASK     0xf000
                /* mask for running head info */                 /* running head
values: */#define        RH_FIRST       0x8000 /* appears on first page */#define
                RH_EVEN  0x4000          /* on even pages */#define      RH_ODD
                0x2000   /* on odd pages */#define   RH_BOTTOM   0x1000      /*
appears on bottom of page */#define      RH_PICT 0x0800       /* this paragraph is a
picture */
```

The `p_flags` field has sometimes the high order bit set (0x8000). The meaning of this could be the same as the setting of this bit in the character formats, where it is also sometimes set. What this bit indicates is unknown.

In the paragraph format run only the tab descriptors needed for explicitely defined tabs are stored. The list of tab descriptors is ended by a tab descriptor with the `t_position` field equal to 0. This final tab descriptor is stored in the file, although this seems not necessary.

A picture is identified by having the `RH_PICT` bit set. In the text part of the document the picture is present, preceded by a 6 byte header. The actual picture data follows after the header. It is encoded in standard PICT format (see tech note #21). The picture header is defined by the structure given below:

```
  /*        This structure defines a picture header*/struct   _phead {/* 00 */
            short    ph_offset; /* offset from left margin *//* 02 */    short
            ph_xdist; /* distortion in x direction *//* 04 */  short ph_ydist;
            /* distortion in y direction */};typedef struct _phead MS_Pict;
```

The `ph_xdist` and `ph_ydist` fields are used to store the distortion of the picture. If both are zero, the picture is undistorted. The exact meaning of the values stored here is unknown.

8. Division blocks.

The next (fifth) part of the file contains the division blocks. One block (128 bytes!) is allocated for every division. The block is filled with the following structure, preceded by a byte count indicating how many bytes are actually stored:

```
  /* *      This structure describes division formats */struct _dformat {/* 00 */
            ushort  d_flags;    /* flags *//* 02 */     ushort      d_pap_len;
            /* total paper length *//* 04 */      ushort      d_pap_wit;  /*
total paper width *//* 06 */        ushortd_p_start;  /* start page # *//* 08 */
            ushort  d_top;      /* top margin *//* 10 */ushort      d_bot;
            /* bottom margin, from top paper *//* 12 */ ushort      d_left;
            /* left margin *//* 14 */ ushort      d_right;    /* right margin,
from left paper *//* 16 */          ushortd_flag_col; /* some flags, number of
columns *//* 18 */     ushort       d_r_top;    /* top run.head pos, from top paper
*//* 20 */     ushort  d_r_bot;    /* bottom run.head pos, from top paper *//* 22 */
            ushort  d_colsp;    /* column spacing *//* 24 */  ushort
            d_gutter;          /* gutter *//* 26 */    ushort      d_pag_top;
            /* page number position from top *//* 28 */ ushort      d_pag_left;
            /* page number position from left *//* 30 */      ushort
            d_unk1;/* 32 */     ushortd_rbot;     /* seems runn. head pos, from
bottom *//* 34 */      short       d_unk2[34];};typedef struct _dformat MS_Div;
```

```
      #define   DFB_MASK            0x00e0/* mask for break */
                /* values for break: */#define  DFB_CONT    0x0000      /* continuous
*/#define       DFB_COL 0x0020      /* column */#define     DFB_PAGE    0x0040
                /* page (default) */#define     DFB_ODD     0x0060      /* odd
*/#define       DFB_EVEN            0x0080/* even */#define DFP_MASK    0x001c
                /* mask for the page # format */                  /* values for page
# format: */#define    DFP_NUM     0x0000/* numeric (1, 2...)*/#define   DFP_ROM
                0x0004  /* roman, upper case (I, II...) */#define   DFP_rom     0x0008
                /* roman, lower case (i, ii...) */#define    DFP_ALF     0x000c
                /* alphabetic, upper case (A, B...) */#define DFP_alf     0x0010
                /* alphabetic, lower case (a, b...) */#define DF_DIV      0x0001
                /* division layout present */

      #define   DEFAULT_PAG         0xffff/* default page number */

      /* mask for flags in the d_flag_col field */                    /* values:
*/#define       DCF_AUTO            0x0200/* auto page numbering on */#define
                DCF_FOOT            0x0100/* 1=footnote at end of division */#define
                DCF_COL 0x00ff      /* mask for number of columns */
```

Some of the values stored in the division blocks seem to have no relation to a division, but rather to the document as a whole. These values are present in all division blocks, but only the value in the first division block is used. The values in the other blocks are just ignored. As usual, all dimensions are in basic units.

The `DF_DIV` bit used to indicate whether the division layout information is stored (`DF_DIV` = `1`) or only the paper dimensions (`DF_DIV` = `0`). If the `DF_DIV` bit is `0`, only 16 bytes of the division block are used.

The division blocks do not contain any information on which part of the text they apply to. This information is stored in the next part of the file, the division list:

9. Division list.

The division list links the division blocks to the text. It consists of division descriptors. These are defined by the structure:

```
      /* *      This structure describes a division descriptor */struct _pdiv {/* 00
*/              ulong   pd_text;         /* where the division starts *//* 04 */
                ushort  pd_unk;          /* unknown *//* 06 */    ulong pd_block;
                /* there is the div block */};typedef struct _pdiv MS_DivD;
```

The `pd_text` field gives the place in the text where the division ends, relative to the start of the text part. Add `0x80` to get the offset from the start of the file. The `pd_block` field gives the offset in the file where the division block starts. If the `pd_block` field is `0xffffffff`, this division has no division block allocated. This seems a division with all default values. The meaning of the `pd_unk` field is unclear.

The structure described in section 8 is present in the corresponding division block, preceded by a bytecount. As with the paragraph and character formats, bytes not stored contain default values. As many division descriptors are in the division list as there are divisions. The division list holds some more information, as can be seen in the structure definition:

```
     /* *      This structure describes the division list */struct _divlist {/* 00
*/              ushort  dl_count;        /* the number of descriptors *//* 02 */
                ushort  dl_unk;          /* some unknown counter *//* 06 */    MS_DivD
                dl_list [0];             /* as many as needed */};typedef struct
_divlist MS_Div;
```

If one block (128 bytes) is not sufficient to store the division list, the list can span block boundaries.

The division list is also the way to se if a `0x0c` character stored within the file is a 'forced new page' ('SHIFT-ENTER') or a 'end of division' character. If there is no entry for the `0x0c` character in the division list, it must be only a new page.

10. Page list.

The last part of the file is the page list. This list contains information about the pages in the document. This information is used to process the "Go To…" command, and to show the small '=' signs in the margin. The page list is updated when the document is repaginated by means of the "Repaginate" command. The items in the page list are described by this structure:

```
     /* *      This structure describes a page list item */struct _page {/* 00 */
                ushort  pg_num;          /* the page number *//* 02 */  ulong
                pg_text;                 /* where it starts */};typedef struct _page
MS_Page;
```

The page numbers seem to be always in numerical order. The `pg_text` field is the offset in the text part where the page starts. Add `0x80` to get the position relative to the start of the file. All page list items are contained in the page list, which fills the last part of the file:

```
     /* *      This structure describes the page list */struct _plist {/* 00 */
                ushort  pl_count;        /* the number  list items *//* 02 */ ushort
                pl_unk; /* some other count *//* 04 */ MS_Page     pl_list [0];
                /* the list */};typedef struct _plist MS_PList;
```

11. End of file.

Just after the page list the end of file is present. The file contains an integral number of MS Word blocks. As a MS Word block is smaller than a physical disk block, the end of file may be in the middle of the last physical disk block.