6. Fenster

Bei Fenstern handelt es sich um rechteckige Bereiche, dargestellt durch zweidimensionale Koordinaten (x, y). Fenster sind Bestandteil von grafischen Benutzeroberflächen und können die Bedienung von Computern ganz erheblich erleichtern. Der Anwender muss sich keine mittels Zeichen einzugebenden Befehle mehr merken oder auswendig lernen. Sondern er kann gezielt den Mauszeiger positionieren (auf ein Symbol "deuten") und mit einem Mausklick Aktionen auslösen.

Bei der Bewegung der Computermaus zählt der Computer ständig die Bewegungsänderungen mit und überträgt diese Änderungen auf die Koordinaten des Mauszeigers, in der Regel ein Pfeil.

Fenster sind in viele einzelne Unterbereiche aufgeteilt, die Symbole, das sind Bilder, (Text)Felder und Schaltflächen, darstellen können. In den meisten Fällen handelt es sich auch bei den Unterbereichen immer um rechteckige Bereiche.

Klickt der Anwender nun auf ein grafisches Symbol (Mausspitze steht darüber), werden die Koordinaten des Mauszeigers mit den Koordinaten der Symbole verglichen und damit das ausgewählte Symbol bestimmt.

RISC OS bietet trotz seines Alters eine der vielleicht bis heute am durchdachtesten, aufgeräumtesten und leistungsfähigsten grafischen Oberflächen überhaupt. Für diese Oberfläche ist RISC OS berühmt und beliebt. Wer die Oberfläche von RISC OS kennt, der kann über die Oberfläche von anderen Systemen eigentlich nur den Kopf schütteln. Und das seit mittlerweile über zwanzig Jahren.

Programme, welche die grafische Benutzeroberfläche von RISC OS nutzen wollen, müssen zwangsläufig im Multitasking laufen. In diesem Modus laufen viele verschiedene Programme *gleichzeitig*. Und ausgerechnet hier spielt RISC OS eine seiner Stärken aus; denn diese Programme sind *kooperativ* und lassen sich in einer Flexibilität miteinander kombinieren wie auf nahezu keinem anderen System.

RISC OS ging von Anfang an seinen eigenen Weg und orientierte sich insbesondere auch an seiner Oberfläche in vielen Punkten an keinem anderen System. Viele Konzepte, die RISC OS schon von Anfang an mitbrachte (z. B. die Symbolleiste, POP-UP-Menüs oder das Ziehen & Fallen lassen) wurden auf anderen Systemen meist erst so nach und nach eingeführt. Und oft bis heute nicht so konsequent und klar für den Anwender umgesetzt wie unter RISC OS.

Für Umsteiger von anderen Systemen stellt dieser Sachverhalt aber auch nach wie vor eine große Schwierigkeit dar, mit der Oberfläche klarzukommen. Weil RISC OS in vielen Dingen einfach so ganz anders funktioniert, als es ein Anwender gewohnt ist. Das fängt schon bei so einfachen und banalen Dingen wie dem Starten oder Beenden von Programmen (RISC OS verfügt über kein Startmenü, und auch das Schließen eines Fensters führt in der Regel nicht dazu, dass ein Programm beendet wird) oder dem Laden und Speichern von Dokumenten an (unter RISC OS muss der Speicherort geöffnet sein, *bevor* man die Dialogbox zum Speichern anwählt. Das mag sich jetzt komisch anhören, hat aber den riesigen Vorteil, dass man den gleichen Speicherort selbst für viele verschiedene Anwendungsprogramme nur einmal öffnen muss).

Die Oberfläche von RISC OS verfolgt wie auch schon bei der zusammen mit RISC OS entstandenen ARM-Rechnerarchitektur ein Minimalprinzip: Mit möglichst wenig Befehlen oder Mausklicks möglichst viel zu erreichen. Dies lässt sich durch geschicktes Kombinieren der Werkzeuge und / oder laufenden Programme in die Tat umsetzen. Die Oberfläche erfordert

jedoch gleichzeitig eine sehr *intensive* Bedienung der Maus: Es werden ständig alle drei Maustasten in Kombination mit der Tastatur verwendet. Wo auch immer möglich, geschieht dies durch Anklicken, Ziehen (*engl. to drag*) und Loslassen (*engl. to drop*) von grafischen Symbolen. Man nimmt also etwas und stellt es einfach wie im richtigen Leben woanders hin. Oder kombiniert es.

Die Programmierung der grafischen Oberfläche von RISC OS ist aus diesem Grund nicht unbedingt einfach. Sie lohnt sich aber, da sie gegenüber anderen Systemen bis heute unheimlich mächtig und anwenderfreundlich ist.

Bild 6.1: Das mit *Listing 6.1* erzeugte Fenster

Erzeugung eines Fensters

Fenster können ganz unterschiedlich aussehen und sich ganz unterschiedlich verhalten. Sie benötigen daher eine große Menge an Informationen (Daten) zur Beschreibung ihrer Eigenschaften.

In der Programmiersprache C und unter Verwendung der C-Bibliothek OSlib wird ein Fenster mittels der Funktion

```
extern wimp_w wimp_create_window (wimp_window const *window);
```

dem System bekannt gemacht. Bei dem Parameter `wimp_window const *window` handelt es sich um einen *Zeiger* auf einen Datenblock. Der Funktion wird also eine Speicheradresse mitgeteilt, ab welcher die für diese Funktion geeignete Daten stehen müssen.

Die Daten müssen eine ganz bestimmte Struktur aufweisen, welche einem fest vom Betriebssystem vorgeschrieben wird. Jedes Bit und jedes Byte dieses Datenblocks haben also in einer festgelegten Reihenfolge eine ganz bestimmte Bedeutung. Die Bedeutung der einzelnen Bits und Bytes kann man den Programmers References Manuals 3-87 (im PDF 97) entnehmen:

Startadresse + 0 Bytes:	*sichtbarer Bereich des Fensters, untere X-Koordinate*
Startadresse + 4 Bytes:	*sichtbarer Bereich des Fensters, untere Y-Koordinate*
Startadresse + 8 Bytes:	*sichtbarer Bereich des Fensters, obere X-Koordinate*
Startadresse + 12 Bytes:	*sichtbarer Bereich des Fensters, Y-Koordinate*
Startadresse + 16 Bytes:	*zum Arbeitsbereich verschobener Inhalt des Fensters, X-Koordinate relativ zum Ursprung des Arbeitsbereichs*

Startadresse + 20 Bytes:	*zum Arbeitsbereich verschobener Inhalt des Fensters, Y-Koordinate relativ zum Ursprung des Arbeitsbereichs*
Startadresse + 24 Bytes:	*Ebene, auf der das Fenster liegen soll (-1 ganz oben, d. h. über allen anderen Fenstern, -2 ganz unten, d. h. hinter allen anderen Fenstern)*
Startadresse + 28 Bytes:	*Fenster-Steuerzeichen (engl. flags)*
Startadresse + 32 Bytes:	*Schriftfarbe der Titelleiste sowie Rahmenfarbe (0xFFu: kein Rahmen)*
Startadresse + 33 Bytes:	*Hintergrundfarbe der Titelleiste*
Startadresse + 34 Bytes:	*Schrift- bzw. Vordergrundfarbe des Arbeitsbereichs*
Startadresse + 35 Bytes:	*Hintergrundfarbe des Arbeitsbereichs*
Startadresse + 36 Bytes:	*Hintergrundfarbe des Rollbalkens*
Startadresse + 37 Bytes:	*Farbe des Schiebers (Rollbalken)*
Startadresse + 38 Bytes:	*Hintergrundfarbe der Titelleiste, wenn Cursor aktiv*
Startadresse + 39Bytes:	*muss immer 0 sein*
Startadresse + 40 Bytes:	*Arbeitsbereich, untere X-Koordinate*
Startadresse + 44 Bytes:	*Arbeitsbereich, untere Y-Koordinate*
Startadresse + 48 Bytes:	*Arbeitsbereich, obere X-Koordinate*
Startadresse + 52 Bytes:	*Arbeitsbereich, obere Y-Koordinate*
Startadresse + 56 Bytes:	*Steuerzeichen des Symbols, welches sich bei jedem Fenster in der Titelzeile befindet*
Startadresse + 60 Bytes:	*Steuerzeichen für den Arbeitsbereich*
Startadresse + 64 Bytes:	*Zeiger auf die Anfangsadresse eines Spritebereichsteuerblocks*[1]
Startadresse + 68 Bytes:	*Minimale Weite des Fensters*
Startadresse + 70 Bytes:	*Minimale Höhe des Fensters*
Startadresse + 72 Bytes:	*Daten der Titelleiste*
Startadresse + 84 Bytes:	*Anzahl der im Fenster enthaltenen Symbole bei erstmaliger Bekanntmachung*
Startadresse + 88 Bytes:	*für jedes zusätzliche Symbol, welches das Fenster enthält, zusätzlich 32 Bytes.*

Auffallend ist, dass hier fast immer mit einem Abstand von 4 Bytes gearbeitet wird. Bei 4 Bytes handelt es sich um 32 Bit. Nun ist die ARM eine 32-Bit-Prozessorarchitektur, d. h. unter einer Speicheradresse können normalerweise immer 32 Bits gleichzeitig angesprochen werden.

Im folgenden sollen zum besseren Verständnis noch einige Parameter genauer erläutert werden.

Koordinaten unter RISC OS

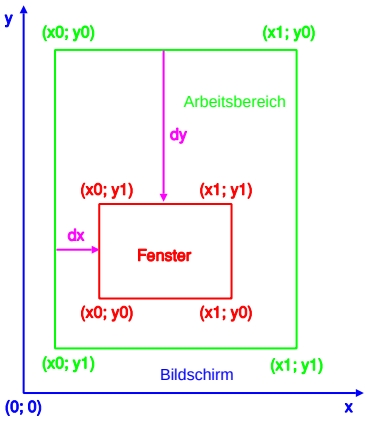
Wie man aus der vorherigen Auflistung des Datenblocks sehen kann, beginnt dieser gleich mit

[1] Ein Sprite (*engl. sprite* u. a. für Geistwesen, Kobold) ist ein kleines Rasterbild, das von der Grafikhardware über das Hintergrundbild bzw. den restlichen Inhalt der Bildschirmanzeige eingeblendet wird. Die Positionierung wird dabei komplett von der Grafikhardware ohne Hilfe des Prozessors durchgeführt. (Quelle: Wikipedia).

Fensterkoordinaten. Diese Koordinaten bedürfen jedoch einer näheren Erläuterung.

RISC OS arbeitet mit OS internen Einheiten (*engl. OS graphics units*). Der Nullpunkt (x, y) hierzu befindet sich in der linken, *unteren* Ecke.

Ein Fenster wird in diesem Koordinatensystem nun durch absolute minimale und maximale (x, y)-Koordinatenpaare beschrieben (Bytes 0 - 12 im Datenblock).



Jedes Fenster verfügt über einen *Arbeitsbereich*. Dieser Arbeitsbereich kann der Fenstergröße entsprechen oder wesentlich größer sein. In letzterem Fall zeigt das Fenster dann nur einen Ausschnitt aus diesem Arbeitsbereich, und mit Hilfe der Schiebebalken kann man dann den sichtbaren Ausschnitt des Arbeitsbereichs (Fensterinhalt) verschieben. Um den Versatz zwischen Arbeitsbereich und Fensterinhalt zu beschreiben, dienen die Bytes 16 (dx) und 20 (dy) des Datenblocks.

Man beachte bei dem Versatz, dass die dy-Komponente mit einem Minuszeichen zu versehen ist, weil bei keinem Versatz (also dx = 0 und dy = 0) Punkt (x0; y1) des Fensters genau in Punkt (x0; y0) des Arbeitsbereichs liegt, aber die y-Koordinate des Koordinatensystems (blau) entgegen gezählt wird. Die dy-Komponente muss damit folglich *negativ* sein.

Steuerzeichen

In diesem Text handelt es sich bei Steuerzeichen um binäre Variablen, die jeweils zwei verschiedene Zustände annehmen können (z. B. an / aus, groß / klein). Im Englischen werden diese als *flags* bezeichnet (von *engl. flag* für Flagge). In anderer deutschsprachiger Literatur findet man hierfür auch häufig den eindeutigen, aus dem Englischen übernommenen Begriff *Flag*.

Weil sich der Autor gegen die Verenglischung der deutschen Sprache sträubt, hat er sich für den (zugegeben nicht ganz eindeutigen) Begriff Steuerzeichen entschieden.

Die Verwendung von OSlib

In C verwendet man anstatt von relativen Speicheradressen (man zählt hierzu den Abstand zu einem festen Wert wie vorangegangen dargestellt) *Datentypen* und *Strukturen* und lässt mit Hilfe dieser Datentypen und Strukturen den Compiler bei der Übersetzung des Programms die Position selbst bestimmen. D. h. der Compiler errechnet sich selbst, an welcher Stelle des Speichers er welchen Wert schreiben muss. Ein C-Programmierer kommt mit tatsächlichen Adressen gar nicht mehr in Berührung. Er muss sich damit nicht mehr die Position der festgelegten Daten, sondern nur noch aussagehfähige Namen merken. Der Compiler bringt dann bei der Übersetzung des Programms in ausführbaren Code die Struktur der Daten in die richtige Reihenfolge.

OSlib bietet uns eine Struktur vom Datentyp `wimp_window` an:

```
typedef struct
{
   os_box visible;
   int xscroll;
   int yscroll;
   wimp_w next;
   wimp_window_flags flags;
   wimp_colour title_fg;
   wimp_colour title_bg;
   wimp_colour work_fg;
   wimp_colour work_bg;
   wimp_colour scroll_outer;
   wimp_colour scroll_inner;
   wimp_colour highlight_bg;
   wimp_extra_window_flags extra_flags;
   os_box extent;
   wimp_icon_flags title_flags;
   wimp_icon_flags work_flags;
   osspriteop_area *sprite_area;
   short xmin;
   short ymin;
   wimp_icon_data title_data;
   int icon_count;
   wimp_icon icons [...];
} wimp_window;
```

Diese Struktur entspricht des in den PRMs dargestellten Aufbaus. Sie sieht nur anders aus. Außerdem ist sie nicht vollständig dargestellt, weil z. B. die Strukturkomponente `os_box visible` weitere Unterkomponenten hat, wie aus dem Stronghelp-Handbuch von OSlib

ersichtlich ist.

Normalerweise wird in C der benötigte Speicher mittels der Deklaration[2] eines Datentypes oder einer Struktur vom Compiler reserviert. So reserviert und weist z. B. die Anweisung

```
int x;
```

2 (freie) Bytes der Variablen `x` zu. Das bedeutet auch, dass hinter *jedem* Datentyp und hinter jeder Struktur der für diesen Datentyp oder Struktur notwendige Speicherbedarf (Größe des Datenblocks) hinterlegt sein muss.

Bei dem für ein Fenster zu reservierenden Datenblock gibt es ein Problem. Wie man an der Zeile

> *Startadresse + 88 Bytes: für jedes zusätzliche Smbol, welches das Fenster enthält, zusätzlich 32 Bytes.*

ersehen kann, variiert der für ein Fenster notwendige Speicherbedarf. In der von OSLib angebotenen Hilfe wird das mittels der Zeile

```
wimp_icon icons [...];
```

verdeutlicht.

Fenster können eine unterschiedliche Anzahl von Symbolen (*engl. icons*) enthalten. Der Speicherbedarf ist also davon abhängig, wie viele zusätzliche Symbole (Schaltflächen, Eingabefelder usw.) ein Fenster enthält. Deshalb kann man den notwendigen Speicher nicht einfach mittels

```
wimp_window const window;
```

reservieren. Das würde nur dann funktionieren, wenn das Fenster *keine* Symbole und damit auch *keine* Inhalte hat. Dann wäre der eingerichtete Datenblock immer gleich groß und passend. Was aber will man mit einem Fenster anfangen, das keine Inhalte hat?

Benötigten Speicher berechnen und reservieren

Deshalb muss man hier die Größe des benötigten Datenblocks anders bestimmen und selbst reservieren.

Um die Größe eines Datentyps zu bestimmen, gibt es in C die Anweisung `sizeof(...)`. Man beachte, dass es sich dabei um eine Anweisung und *nicht* um eine Funktion handelt! `sizeof(...)` ist in C so elementar, so grundlegend, dass sie jedem C-Compiler ohne jede zusätzliche Bibliothek bekannt ist.

Mittels den Anweisungen

[2] Erklärung, Bekanntmachung gegenüber dem Compiler

```
sizeof(wimp_window) + i * sizeof(wimp_icon)                (1)
```

kann man nun die erforderliche Größe des Datenblocks für ein Fenster berechnen, wobei `i` für die Anzahl der in einem Fenster enthaltenen Symbole steht.

Den Speicher reserviert man dann mittels der C-Funktion `malloc (size_t size)`, welche in der Bibliothek `stdlib.h` enthalten ist. Den reservierten Speicher muss man anschließend natürlich noch einem Zeiger vom Datentyp `wimp_window` zuordnen, denn die Funktion

```
extern wimp_w wimp_create_window (wimp_window const *window);
```

erwartet ja gerade eben einen solchen Zeiger als einzigen Parameter.

Mal angenommen, man wolle ein Fenster erzeugen, welches nur ein einziges Symbol enthälte. Dann würde der für den Datenblock notwendige C-Code folgendermaßen aussehen:

```
wimp_window *window;
window = malloc(sizeof(wimp_window) + 1*sizeof(wimp_icon));
```

Leider kann C *nicht* die Größe von Datenfeldern (*engl. arrays*) überprüfen. Wenn man den Datenblock für ein Fenster mittels vorstehendem Zweizeiler eingerichtet hat, lässt sich immer noch die Anweisung

```
window->icons[2].data.... = ...;
```

ausführen, ohne dass der Compiler bei der Übersetzung des Programms eine Fehlermeldung bringen und die Übersetzung abbrechen würde. (Die drei Punkte stehen hier als Platzhalter für noch erforderlichen Code.) Mittels `window->icons[2].data... = ...;` wird aber ein Speicherbereich überschrieben, der nicht mehr zu dem für das Fenster reservierten Datenblock gehört, sondern eigentlich *anderweitig* genutzt wird und damit zu einem Programmfehler oder unerwünschten Nebenwirkungen führen kann.

Parameter eintragen

In C werden Werte mit Hilfe des Punktoperators `.` in Strukturkomponenten eingetragen. Weil es sich bei `windows` jedoch um einen *Zeiger* handelt, müssen wir `windows` den Sternoperator zur Dereferenzierung voranstellen: `*windows` erlaubt damit einen Zugriff auf den Inhalt der Speicheradresse, deren Wert in im Zeiger `windows` hinterlegt ist (`windows = ...` würde den *Wert* der Adresse, also die Stelle ändern, auf die der Zeiger `windows` zeigt, nicht aber deren Inhalt). Nun wird in C jedoch der Punktoperator grundsätzlich *vor* dem Sternoperator behandelt. Die Anweisung

```
*window.visible.x0 = 200;
```

würde damit so interpretiert werden, dass in der Strukturkomponente window.visible.x0 eine Speicheradresse hinterlegt ist, an deren Inhalt die Zahl 200 geschrieben werden soll. Das ist jedoch falsch und kann nicht funktionieren, weil es sich nur bei `windows` um einen Zeiger handelt und `x0` vom Datentyp Ganzzahl ist. Wir müssen daher Klammern setzen:

```
(*window).visible.x0 = 200;
```

wird vom Compiler richtig verstanden. Statt den Klammern sollte jedoch besser der eigens dafür eingeführte Pfeil-Operator `->` zur Auswahl einer Struktur-Komponente über einen Zeiger verwendet werden:

```
window->visible.x0 = 200;
```

Diese Anweisung setzt nun den Wert der ersten vier Bytes des eingerichteten Datenblocks auf den Ganzzahl-Wert 200. Dabei handelt es sich *nicht* um die ASCII-Zeichenfolge 200, sondern um die *binäre* Darstellung des Wertes 200 (Darstellung von 200 im binären Zahlensystem: 0000 0000 1100 1000).

Einzelne Bits setzen

Ganz interessant wird jetzt der Eintrag von `wimp_window_flags flags`. Hier müssen einzelne Bits gesetzt werden, die als Steuerzeichen (*engl. flags*) dienen (an / aus, an / aus, an / aus...). Die Bedeutung der einzelnen Bits können ebenfalls den PRMs, Seite 3-88 - 3-98 entommen werden (im PDF Seiten 98 und 99):

Bit	**Bedeutung**
0	Fenster hat eine Titelzeile - Bit sollte nicht mehr genutzt werden
1	Fenster kann verschoben werden
2	Fenster hat einen vertikalen Rollbalken - Bit sollte nicht mehr genutzt werden
3	Fenster hat einen horizontalen Rollbalken - Bit sollte nicht mehr genutzt werden
4	das System kann den Fensterinhalt ohne Hilfe des Tasks aktualisieren
5	Fenster stellt einen Werkzeugkasten dar
6	Fenster kann über die sichtbaren Oberfläche hinausgeschoben werden
7	Fenster hat keine Schließ- oder Nach-Hinten-Symbole - Bit sollte nicht mehr genutzt werden
8	Roll- und Pfeiltasten aktiviert
9	Roll- und Pfeiltasten aktiviert, aber ohne automatische Wiederholung bei dauerhaft gedrückter Maustaste
10	Fensterfarben werden als GCOL anstatt von Wimp-Farben interpretiert
11	kein anderes Fenster hinter diesem erlaubt
12	Der Druck auf Spezialtasten wird mitgeteilt, wenn das Fenster geöffnet ist.
13	zwingt Fenster dazu, auf dem Bildschirm zu bleiben
14	Fenster lässt sich nicht nach rechts aufziehen (wird unterdrückt)

15 Fenster lässt sich nicht nach unten aufziehen (wird unterdrückt)

16 - 20 werden nicht vom Anwender, sondern von RISC OS gesetzt, und haben folgende Bedeutungen:

- 16 Fenster ist geöffnet
- 17 Fenster ist vollständig sichtbar, d. h. dass keine Teile von ihm irgendwie verdeckt werden
- 18 das Fenster wurde mittels dem Größenumschalter auf maximale Größe umgeschaltet
- 19 das aktuelle Ereignis Open_Windows_Request wurde durch den Größenumschalter herbeigeführt
- 20 das Fenster verfügt über den Eingabecursor; es ist für die Eingabe mittels Tastatur aktiv

21 beim nächsten Ereignis Open_Window wird das Fenster auf den Bildschirm gezaubert; dabei wird dieses Bit gelöscht. Dieses Bit wird bei verschiedenen Ereignissen auch von RISC OS selbst gesetzt.

22 - 23 reserviert, müssen auf 0 gesetzt sein

24 Fenster verfügt über ein Nach-Hinten-Symbol

25 Fenster verfügt über ein Schließ-Dich-Symbol

26 Fenster hat eine Titelleiste

27 Fenster hat einen Größenumschalter

28 Fenster hat einen vertikalen Rollbalken

29 Fenster hat das Symbol zur Veränderung seiner Größe

30 Fenster hat einen horizontalen Rollbalken

31 falls gesetzt, werden Bits 24 - 30 benutzt, um die Kontrollsymbole eines Fensters zu steuern, andernfalls werden hierfür Bits 0, 2, 3 und 7 verwendet.

Man könnte nun den Wert für die Fenster-Steuerzeichen von Hand berechnen oder mittels dem Programm Bits wie in Teil "Multitasking und Pollschleife" dieser Serie dargestellt. In C gibt es jedoch eine andere, vielleicht bessere Methode: Man gibt direkt die einzelnen Bits an und lässt den Compiler den Wert selbst berechnen. Das geht, indem man aussagefähige Konstanten verwendet, die uns von OSlib zur Verfügung gestellt werden. Ein Blick in das StrongHelp-Handbuch von OSLib offenbart uns:

Konstante	**hexdezimale Darstellung**	**binäre Darstellung (Bit jeweils sichtbar)**
wimp_WINDOW_MOVEABLE	0x0000 0002u	0000 0000 0000 0000 0000 0000 0000 0010
wimp_WINDOW_AUTO_REDRAW	0x0000 0010u	0000 0000 0000 0000 0000 0000 0001 0000
wimp_WINDOW_PANE	0x0000 0020u	0000 0000 0000 0000 0000 0000 0010 0000
wimp_WINDOW_NO_BOUNDS	0x0000 0040u	0000 0000 0000 0000 0000 0000 0100 0000
wimp_WINDOW_SCROLL_REPEAT	0x0000 0100u	0000 0000 0000 0000 0000 0001 0000 0000
wimp_WINDOW_SCROLL	0x0000 0200u	0000 0000 0000 0000 0000 0010 0000 0000
wimp_WINDOW_REAL_COLOURS	0x0000 0400u	0000 0000 0000 0000 0000 0100 0000 0000
wimp_WINDOW_BACK	0x0000 0800u	0000 0000 0000 0000 0000 1000 0000 0000
wimp_WINDOW_HOT_KEYS	0x0000 1000u	0000 0000 0000 0000 0001 0000 0000 0000
wimp_WINDOW_BOUNDED	0x0000 2000u	0000 0000 0000 0000 0010 0000 0000 0000
wimp_WINDOW_IGNORE_XEXTENT	0x0000 4000u	0000 0000 0000 0000 0100 0000 0000 0000
wimp_WINDOW_IGNORE_YEXTENT	0x0000 8000u	0000 0000 0000 0000 1000 0000 0000 0000
wimp_WINDOW_OPEN	0x0001 0000u	0000 0000 0000 0001 0000 0000 0000 0000
wimp_WINDOW_NOT_COVERED	0x0002 0000u	0000 0000 0000 0010 0000 0000 0000 0000
wimp_WINDOW_FULL_SIZE	0x0004 0000u	0000 0000 0000 0100 0000 0000 0000 0000

```
wimp_WINDOW_TOGGLED           0x0008 0000u          0000 0000 0000 1000 0000 0000 0000 0000
wimp_WINDOW_HAS_FOCUS         0x0010 0000u          0000 0000 0001 0000 0000 0000 0000 0000
wimp_WINDOW_BOUNDED_ONCE      0x0020 0000u          0000 0000 0010 0000 0000 0000 0000 0000
wimp_WINDOW_PARTIAL_SIZE      0x0040 0000u          0000 0000 0100 0000 0000 0000 0000 0000
wimp_WINDOW_FURNITURE_WINDOW  0x0080 0000u          0000 0000 1000 0000 0000 0000 0000 0000
wimp_WINDOW_FOREGROUND_WINDOW 0x0080 0000u          0000 0000 1000 0000 0000 0000 0000 0000
wimp_WINDOW_BACK_ICON         0x0100 0000u          0000 0001 0000 0000 0000 0000 0000 0000
wimp_WINDOW_CLOSE_ICON        0x0200 0000u          0000 0010 0000 0000 0000 0000 0000 0000
wimp_WINDOW_TITLE_ICON        0x0400 0000u          0000 0100 0000 0000 0000 0000 0000 0000
wimp_WINDOW_TOGGLE_ICON       0x0800 0000u          0000 1000 0000 0000 0000 0000 0000 0000
wimp_WINDOW_VSCROLL           0x1000 0000u          0001 0000 0000 0000 0000 0000 0000 0000
wimp_WINDOW_SIZE_ICON         0x2000 0000u          0010 0000 0000 0000 0000 0000 0000 0000
wimp_WINDOW_HSCROLL           0x4000 0000u          0100 0000 0000 0000 0000 0000 0000 0000
wimp_WINDOW_NEW_FORMAT        0x8000 0000u          1000 0000 0000 0000 0000 0000 0000 0000
```

Hinter all diesen Variablen verstecken sich Werte. Ganz wichtig ist der Binärwert (binäre Darstellung des Inhalts der Konstante). Denn erst an dieser sehen wir Klartext und können damit verstehen, was in Wirklichkeit geschieht.

Was jetzt in Verbindung mit solchen Konstanten genutzt werden muss, das sind die bitweisen Operatoren UND (&), ODER (|) sowie die beiden Schiebeoperatoren << und >>.

Um das zu verstehen, müssen wir in die Boole'sche Algebra wechseln. UND sowie ODER sind folgendermaßen definiert:

a	b	a & b (a UND b)	a | b (a ODER b)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Wollen wir ein einziges Steuerzeichen / Bit setzen und sollen alle anderen aus (d. h. 0) sein, reicht dessen Eintrag an der festgelegten Stelle im Datenblock:

```
window->flags = wimp_WINDOW_MOVEABLE;
```

Wollen wir ein weiteres Steuerzeichen / Bit dazuschalten, müssen wir dieses mit dem ersteren verknüpfen. Denn sonst würde ja das erstere Bit wieder gelöscht, d. h. auf null gesetzt werden (da von der neuen Konstante überschrieben). Sinnvollerweise müssen wir hierzu den Operator | (ODER) verwenden, wie man aus vorherstehender Tabelle sehen kann. 0 und 1 müssen stets 1 ergeben. Denn sonst würde das vorherige Bit ebenfalls gelöscht:

```
window->flags = wimp_WINDOW_MOVEABLE | wimp_WINDOW_AUTO_REDRAW;
```

Auf diese Art kann man nun alle Steuerzeichen für die Fenster setzen:

```
window->flags = wimp_WINDOW_MOVEABLE | wimp_WINDOW_AUTO_REDRAW |
   wimp_WINDOW_BACK_ICON | wimp_WINDOW_CLOSE_ICON |
```

```
wimp_WINDOW_TITLE_ICON | wimp_WINDOW_TOGGLE_ICON |
wimp_WINDOW_VSCROLL | wimp_WINDOW_SIZE_ICON |
wimp_WINDOW_HSCROLL | wimp_WINDOW_NEW_FORMAT;
```

Bei einigen Konstanten von OSlib ist es notwendig, *vor* der Anwendung deren Position zu verschieben. Das steht aber grundsätzlich im Stronghelp-Handbuch von OSlib mit dabei.

Sehen wir uns doch einmal die Bedeutung der Steuerzeichen für den Arbeitsbereich eines Fensters (Startadresse + 60 Bytes bzw. `work_flags`) an. Hier haben nur die Bits 12 bis 15 eine Bedeutung:

Wert (dezimal)	**Bedeutung**
0	Ignoriere alle Mausklicks
1	Mauszeiger steht über dem Fenster
2	Mausklick (mit automatischer Wiederholung)
3	Mausklick (wird hier nur ein einziges Mal gemeldet)
4	Loslassen der Maustaste im Arbeitsbereich eines Fensters benachrichtigt die Aufgabe / das Programm
5	Doppelklick informiert Aufgabe / Programm
6	Wie 3, Anwender kann aber auch mit der Maus "ziehen" (Tastenzustand *16)
7	Wie 4, Anwender kann aber auch mit der Maus "ziehen" (Tastenzustand * 16)
8	Wie 5, Anwender kann aber auch mit der Maus "ziehen" (Tastenzustand * 16)
9	Wie 3
10	Mausklick liefert Tastenzustand * 256 Ziehen liefert Tastenzustand * 16 Doppelklick liefert Tastenzustand *1
11	Mausklick liefert Tastenzustand Ziehen liefert Tastenzustand *16
12 - 14	reserviert
15	Mausklick aktiviert Schreibmarke; Fenster wird damit aktiv (Titelleiste wird hervorgehoben)

Diese Steuerzeichen sagen RISC OS, welches Ereignis es an das Programm melden soll, wenn der Mauszeiger innerhalb des Arbeitsbereichs eines Fensters steht und eine Maustaste gedrückt wird. Wichtig an dieser Stelle zu verstehen ist, dass hier *nicht* jedes Bit seine eigene Bedeutung hat, sondern immer nur der Gesamtwert von allen vier Bits gesehen werden muss.

In OSlib existieren für diese Werte die Konstanten wimp_BUTTON_...:

Konstante	**Hexwert**	**Binärwert**
`wimp_BUTTON_NEVER`	`0x0u`	`0000`
`wimp_BUTTON_ALWAYS`	`0x1u`	`0001`

```
wimp_BUTTON_REPEAT            0x2u                0010
wimp_BUTTON_CLICK             0x3u                0011
wimp_BUTTON_RELEASE           0x4u                0100
wimp_BUTTON_DOUBLE_CLICK      0x5u                0101
wimp_BUTTON_CLICK_DRAG        0x6u                0110
wimp_BUTTON_RELEASE_DRAG      0x7u                0111
wimp_BUTTON_DOUBLE_DRAG       0X8u                1000
wimp_BUTTON_MENU_ICON         0x9u                1001
wimp_BUTTON_DOUBLE_CLICK_DRAG 0xAu                1010
wimp_BUTTON_RADIO             0xBu                1011
wimp_BUTTON_WRITE_CLICK_DRAG  0xEu                1110
wimp_BUTTON_WRITEABLE         0xFu                1111
```

Wollen wir diese Konstanten anwenden, so haben wir grundsätzlich ein Problem. Die Anweisung

```
window->work_flags = wimp_BUTTON_CLICK;
```

ist falsch. Warum? Weil `wimp_BUTTON_CLICK` nicht die Bits 12 bis 15, sondern die Bits 0 bis 3 setzt:

```
wimp_BUTTON_ALWAYS = 0000 0000 0000 0000 0000 0000 0000 0010
```

Der Wert in `work_flags` müsste aber lauten:

```
work_flags = 0000 0000 0000 0000 0010 0000 0000 0000
```

(Jeweils binäre Darstellung.) Die Bits 12 - 15 sind blau hervorgehoben.

Der Wert von `wimp_BUTTON_CLICK` müsste also um 12 Bits nach links geschoben werden. Dies können wir mit dem Schiebeoperator `<<` von C realisieren:

```
work_flags = wimp_BUTTON_CLICK << 12;
```

Wie in C üblich, muss man sich die 12 aber auch nicht merken. sondern wird ebenfalls durch eine Konstante ersetzt. Diese Konstante findet man im StrongHelp-Handbuch von OSlib unter jedem Eintrag zu `wimp_BUTTON_...` und heißt `wimp_ICON_BUTTON_TYPE_SHIFT`. Damit gelangen wir zu:

```
work_flags = wimp_BUTTON_CLICK << wimp_ICON_BUTTON_TYPE_SHIFT;
```

Mit diesem ganzen Wissen können wir jetzt den kompletten Datenblock einrichten, der unter RISC OS zur Beschreibung eines Fenster notwendig ist:

```
wimp_window *window;
```

```
window = malloc(sizeof(wimp_window) + 1*sizeof(wimp_icon));
window->visible.x0 = 200;
window->visible.y0 = 200;
window->visible.x1 = 400;
window->visible.y1 = 400;
window->xscroll = 0;
window->yscroll = 0;
window->next = wimp_TOP;
window->flags = wimp_WINDOW_MOVEABLE | wimp_WINDOW_AUTO_REDRAW |
     wimp_WINDOW_BACK_ICON | wimp_WINDOW_CLOSE_ICON |
     wimp_WINDOW_TITLE_ICON | wimp_WINDOW_TOGGLE_ICON |
     wimp_WINDOW_VSCROLL | wimp_WINDOW_SIZE_ICON |
     wimp_WINDOW_HSCROLL | wimp_WINDOW_NEW_FORMAT;
window->title_fg = wimp_COLOUR_BLACK;
window->title_bg = wimp_COLOUR_LIGHT_GREY;
window->work_fg =  wimp_COLOUR_BLACK;
window->work_bg = wimp_COLOUR_VERY_LIGHT_GREY;
window->scroll_outer = wimp_COLOUR_MID_LIGHT_GREY;
window->scroll_inner = wimp_COLOUR_VERY_LIGHT_GREY;
window->highlight_bg = wimp_COLOUR_CREAM;
window->extent.x0 = 0;
window->extent.y0 = -600;
window->extent.x1 = 600;
window->extent.y1 = 0;
window->title_flags = wimp_ICON_TEXT | wimp_ICON_HCENTRED |
     wimp_ICON_VCENTRED;
window->work_flags = wimp_BUTTON_CLICK <<
     wimp_ICON_BUTTON_TYPE_SHIFT;
window->sprite_area = 0;
strncpy (window->title_data.text, "Hello world!", 13);
window->icon_count = 0;
window->xmin = 0;
window->ymin = 0;
```

Wie man sehen kann, enthält unser Programm keine Symbole (*engl. icons*). Es ist leer. Sinnvollerweise sollte das Fenster aber Symbole enthalten. Wir ersparen uns das an dieser Stelle zur Übersichtlichkeit jedoch noch, um nicht zuviel auf einmal zu machen.

Die Funktion `extern wimp_w wimp_create_window (wimp_window const * window);` liefert einen Wert zurück, der speziell für das Fenster vergeben worden ist (*engl. window handle*). Alle Fenster aller laufenden Programme bekommen einen bestimmten Wert zugeschrieben. Über diese Werte wird jedes Fenster gesteuert und angesprochen.

Bevor wir das Fenster erzeugen können, brauchen wir deshalb noch eine Variable vom Typ `wimp_w,` in welcher der von RISC OS vergebene Wert hinterlegt wird. Diese Variable ist in der Struktur vom Typ `wimp_block` bereits enthalten:

```
wimp_block block;
```

Jetzt endlich können wir das Fenster mittels der Funktion `wimp_create_window(...)` einrichten:

```
block.open.w = wimp_create_window (window);
```

Damit ist das Fenster zwar eingerichtet. RISC OS weiß jetzt über unser Fenster Bescheid. Aber sehen wird man so noch immer nichts. Erst eine weitere Funktion lässt unser Fenster auf dem Bildschirm erscheinen, nämlich `extern void wimp_open_window(wimp_open * open)`.

`wimp_open_window`(...) benötigt jedoch noch einige zusätzliche Parameter in `block.open`, die wir zuvor noch eintragen müssen:

`block.open.visible.x0`	sichtbarer Bereich, minimale x-Koordinaten-Komponente
`block.open.visible.y0`	sichtbarer Bereich, minimale y-Koordinaten-Komponente
`block.open.visible.x1`	sichtbarer Bereich, maximale x-Koordinaten-Komponente
`block.open.visible.y1`	sichtbarer Bereich, maximale y-Koordinaten-Komponente
`block.open.xscroll`	sichtbarer Ausschnitt, relativ zum Ursprung des Arbeitsbereichs
`block.open.yscroll`	sichtbarer Ausschnitt, relativ zum Ursprung des Arbeitsbereichs
`block.open.next`	gibt an, auf welcher Ebene das Fenster geöffnet werden soll (-1: ganz oben, -2: ganz unten etc.)

Vergleicht man die gerade eben aufgeführten Parameter in `block` mit dem Datenblock `*windows` unseres Fensters, so stellt man fest, dass alle eben gerade genannten Parameter ebenfalls im Datenblock `*windows` unseres Fensters zu finden sind. Warum ist das so?

Wenn wir ein Fenster erstmalig öffnen (sichtbar machen), greifen wir in der Regel auf die Werte zurück, welche im Datenblock `*windows` unseres Fensters hinterlegt sind. Denn Fenster sollen beim Öffnen (Erscheinen auf dem Bildschirm) immer an der gleichen Position erscheinen.

Wenn wir aber ein Fenster verschieben, dessen Größe oder Ebene verändern, muss sich RISC OS dessen neue Position merken. Genau dies geschieht in `block.open`. Beim Verändern eines Fensters werden folglich die Werte in `block.open` überschrieben. Die ursprünglichen Werte in `*windows` aber bleiben unangetastet!

Es ist also eine Grundeinstellung für das Fenster im Datenblock `*windows` vorhanden, welche beim erstmaligen Zeichnen (Öffnen) eines Fensters auf dem Bildschirm als Vorlage dient. Beim Verändern eines Fensters aber müssen diese Werte durch andere Werte überschrieben werden.

Man kopiert jetzt beim erstmaligen Zeichnen (Öffnen) eines Fensters für gewöhnlich die Werte

von `*windows` nach `block.open`:

```
 block.open.visible.x0 = window->visible.x0;
 block.open.visible.y0 = window->visible.y0;
 block.open.visible.x1 = window->visible.x1;
 block.open.visible.y1 = window->visible.y1;
 block.open.xscroll = window->xscroll;
 block.open.yscroll = window->yscroll;
 block.open.next = window->next;
```

Jetzt erst können wir das Fenster sichtbar machen:

```
wimp_open_window (&(block.open));
```

Wenn wir jetzt das Fenster mit der Maus verändern wollen, überschreibt RISC OS die Werte in `block.open` und teilt das Ereignis `wimp_OPEN_WINDOW_REQUEST` unserem Programm mit. Wir müssen daraufhin mit dem Neuzeichnen des Fensters reagieren:

```
        case wimp_OPEN_WINDOW_REQUEST:
                wimp_open_window(&(block.open));
                break;
```

Reagiert man nicht auf das Ereignis `wimp_OPEN_WINDOW_REQUEST`, so bleibt das Fenster unverändert auf dem Schirm. Es lässt sich dann weder ziehen, noch die Größe verändern oder der Fensterinhalt mit Hilfe der Rollleisten verschieben.

Möchte man das Fenster mit Hilfe der Kreuz-Schaltfläche schließen können, so muss man auf ein weiteres Ereignis, nämlich auf das Ereignis `wimp_CLOSE_WINDOW_REQUEST`, reagieren:

```
        case wimp_CLOSE_WINDOW_REQUEST:
                wimp_close_window(block.close.w);
                quit_pending = true;
                break;
```

Im folgenden zur besseren Übersichtlichkeit noch einmal das vollständige Programmlisting, welches ein Multitasking-Programm (Aufgabe) unter RISC OS anmeldet und ein Fenster auf den Bildschirm zeichnet:

```
#include "oslib/wimp.h"
#include <stdbool.h>
#include <stdio.h>

int main()
{

 // initialise task
 wimp_version_no version_out;
```

```
wimp_t task_handle;
task_handle=wimp_initialise(310, "Hello World!", NULL, &
    version_out);

// richte Datenblock für Fenster ein
wimp_window *window;
window = malloc(sizeof(wimp_window) + 1*sizeof(wimp_icon));
window->visible.x0 = 200;
window->visible.y0 = 200;
window->visible.x1 = 400;
window->visible.y1 = 400;
window->xscroll = 0;
window->yscroll = 0;
window->next = wimp_TOP;
window->flags = wimp_WINDOW_MOVEABLE | wimp_WINDOW_AUTO_REDRAW
    | wimp_WINDOW_BACK_ICON | wimp_WINDOW_CLOSE_ICON |
    wimp_WINDOW_TITLE_ICON | wimp_WINDOW_TOGGLE_ICON |
    wimp_WINDOW_VSCROLL | wimp_WINDOW_SIZE_ICON |
    wimp_WINDOW_HSCROLL | wimp_WINDOW_NEW_FORMAT;
window->title_fg = wimp_COLOUR_BLACK;
window->title_bg = wimp_COLOUR_LIGHT_GREY;
window->work_fg =  wimp_COLOUR_BLACK;
window->work_bg = wimp_COLOUR_VERY_LIGHT_GREY;
window->scroll_outer = wimp_COLOUR_MID_LIGHT_GREY;
window->scroll_inner = wimp_COLOUR_VERY_LIGHT_GREY;
window->highlight_bg = wimp_COLOUR_CREAM;
window->extent.x0 = 0;
window->extent.y0 = -600;
window->extent.x1 = 600;
window->extent.y1 = 0;
window->title_flags = wimp_ICON_TEXT | wimp_ICON_HCENTRED |
    wimp_ICON_VCENTRED;
window->work_flags = wimp_BUTTON_CLICK <<
    wimp_ICON_BUTTON_TYPE_SHIFT;
window->sprite_area = 0;
strncpy (window->title_data.text, "Hello world!", 13);
window->icon_count = 0;
window->xmin = 0;
window->ymin = 0;
wimp_block block;
block.open.w = wimp_create_window (window);
block.open.visible.x0 = window->visible.x0;
block.open.visible.y0 = window->visible.y0;
block.open.visible.x1 = window->visible.x1;
block.open.visible.y1 = window->visible.y1;
block.open.xscroll = window->xscroll;
block.open.yscroll = window->yscroll;
block.open.next = window->next;
```

```
 // erzeuge Fenster
 wimp_open_window (&(block.open));

 // Variablen für's Pollen
 wimp_event_no event;
 wimp_poll_flags mask=1;
 osbool quit_pending = FALSE;

 // Zentrale Poll-Schleife
 while (!quit_pending)
    {
      event = wimp_poll(mask, &block, NULL);
      switch (event)
      {
       case wimp_OPEN_WINDOW_REQUEST:
                wimp_open_window(&(block.open));
                break;

       case wimp_CLOSE_WINDOW_REQUEST:
                wimp_close_window(block.close.w);
                quit_pending = true;
                break;

        case wimp_USER_MESSAGE:
        case wimp_USER_MESSAGE_RECORDED:
           if (block.message.action == message_QUIT)
             {
                 quit_pending = true;
                 break;
             }
       }

    }
 wimp_close_down(task_handle);
 return 0;
}
```

Listing 6.1 - Erzeugung eines Fensters