Getting started with IDE68K

Creating a new file

Usually the first thing to do when you start writing a program, is to create a new file for the program. In IDE68K you can create a new file with item **New** from the **File** menu, the key combination Ctrl + N or the [ ] button.

Select **Assembly program** from the dialog box. A file with the temporary name *Newfile1.asm* will be created, IDE68K will ask for a new name when a file with that name is saved. Optionally, standard text from a template file can be inserted every time a new file is created. The name of the template file can be specified with the **Templates** item from the **Options** menu. There are two possibilities, one for a new Assembly file and another for a new C file.

Opening an existing file

The easiest way however to learn IDE68K is to assemble and run a number of example programs. These programs can be found in the directory *C:\Ide68k\Examples*.

The first program to assemble and run is *Hello68k.asm*. You can load it in the IDE68K editor by selecting command **Open** from the **File** menu, the key combination Ctrl + O or the [ ] button.

Open *Hello68k (.asm)* from the "Open File" dialog.

```
LF        equ     $0A            ; ASCII code Line Feed
CR        equ     $0D            ; ASCII code Carriage Return

          org     $400           ; start of program (>= $400)

          lea     text,A0        ; get address of text string in A0
          trap    #15
          dc.w    7              ; system call 7 (print string)
          trap    #15
          dc.w    0              ; system call 0 (exit program)

; string to print, string ends with NULL
text      dc.b    'Hello IDE68K!',CR,LF,0
```

Figure 1. *Hello68k.asm*

It is possible that this program is not listed in the dialog. Most likely this is because the example programs were installed in another disk and/or directory. Try to find the program in the other directory and make it the default. Select item **Directories** from the **Options** menu to specify the correct default directory. Entries for library and include directories may be specified as well. The library directories are searched for files specified by the Include instruction in Assembly programs, the Include directories by the #include statement in C programs. Select **Save options** to make the changes permanent.

Assemble the program by selecting command **Build all** from the **Project** menu, the F7 key or the ⚒ button.

A window will open to indicate that *C:\Ide68k\Examples\Hello68k.asm* is being assembled and that there are no errors. Close this window by typing "Enter" or clicking the "OK" button.

Open the file *Hello68k.lst* in the editor to show the assembled instructions.

Run the program with the Command-line Simulator. Select item **Simulator** from the **Run** menu, type the F9 key or click the ▣ button.

Make sure Hello68k.asm is in the top window of the Editor.

A black window of 25 lines high and 80 characters wide is displayed. This is the Command-line Simulator. It is expecting commands to be typed in.

Type H (+ Enter) to show a list of available commands.

If the simulator is started from within the IDE68K editor, the assembled program is already loaded in memory. Run the program with the command G (+ Enter).

Try other commands like DR (Display Registers), DM 400 41F (Display Memory), DA 400 41F (DisAssemble) and others.

Hello68k.asm makes use of a so-called System Call to print the string. In IDE68K a system call is made by the TRAP #15 instruction followed by a 16-bit word to indicate the desired function. There are about 20 different system calls in IDE68K. See the on-line Help system for a full explanation of them.

Change the program slightly by typing a different string. Run the program directly in the simulator (without assembling first). IDE68K will notice the program has changed and assemble it before running the simulator. The modification is only with the text in the editor-window, it is not applied to the file itself. Use the **Save** command from the **File** menu, the key-combination CTRL + S or the 💾 button to do so.

Introduce an error in the program, for instance by changing the "lea"-instruction to "leax". Notice the syntax-coloring parser will not recognize this instruction.

Try to assemble the program either by the "Build all" command or by directly running the simulator. A window will appear telling *Hello68k.asm* is being assembled and an error is found. Click on the "Errors" button to show what is causing the error. Double-clicking on the error in the Errors-window will activate the editor and the line with the error is marked. A description of the error is in the status bar at the bottom of the window. Correct the error and immediately assemble the program again.

Changing the "lea"-instruction to "lea.b" results in a Warning message. (A "lea"-instruction as always long, the assembler knows that and ignores the .b). To correct, follow the same procedure as for errors.

Context-sensitive help

One of the features of IDE68K is the availability of context-sensitive online help for a specific keyword of the program. These keywords include mnemonics and directives for the Assembler and standard library functions for the C compiler.

To get online help information, move the *caret* for instance to the "lea" instruction in the program. (The caret or "keyboard cursor" is the small vertical line indicating where a character entered on the keyboard is inserted in the text. You can move the caret with the up/down and left/right arrow keys or synchronize the caret with the mouse cursor by clicking the left mousebutton.)

Typing the F1 function key on the keyboard opens a window showing essential information for the "lea"-instruction.

Move the caret to the "org"-directive and type the F1 function key to get information for this directive. Try other mnemonics and directives in the program.

More assembly programs.

Load *Bounce.asm* in the editor. This is a small assembly program which prints so-called ANSI-escape strings to the terminal in order to control the position of the cursor.

The cursor is moved one position up and right or one position down and left from the current position, the direction is changed every time the cursor is at the edge of the screen. The result is that the cursor seems to "bounce" across the screen.

Try other programs from the Examples directory such as *Histogram.asm*, *Tbi68k.asm* and *Encrypt.asm*

Histogram.asm asks the user for a line of text and displays the number of times that a letter is found in a histogram-like format. The letter E should have the highest frequency in standard English text.

Tbi68k.asm is a BASIC interpreter written entirely in assembly language. BASIC stands for Beginners All purpose Symbolic Instruction Code It is a very simple programming language to teach computer programming. It was Bill Gates himself who demonstrated that a high-level language such as BASIC could be run on "shoebox" computers of the mid-1970's. That was the start of Microsoft…

Run *Tbi68k.asm* and after the BASIC prompt enter the program

```
10 for i = 1 to 20
20 print i * i
30 next i
40 stop
```

Figure 2. *A small BASIC program*

Type the command LIST to show the program and RUN to run it. This will print the squares of 1 to 20 on the screen. The BASIC program cannot be saved.

Encrypt.asm encrypts a given string (the *plaintext*) with a second string (the *key*). When the encrypted string is processed with the same key, the result is the original plaintext. (Unfortunately, when the encrypted text is processed with the plaintext, the result is the key itself).

The program makes use of so-called macro's in assembly language. These are predefined sequences of instructions that are inserted (*expanded*) in the program when the name of the macro appears in the opcode field. Notice the way in which arguments are passed. A macro is defined with formal parameters which are substituted with actual arguments when the macro is expanded. This is similar to the way C handles #define macro arguments. See *encrypt.lst* for the result.

A command-line interface was used with MS-DOS computers and is still favored by many system administrators on LINUX (UNIX) systems. It often is the native interface to many small or embedded computer systems.

The Visual Simulator

The so-called Visual Simulator is a 68000 simulator with a Graphical User Interface. It uses exactly the same simulator "engine" as the Command-line Simulator but instead of typing commands like DR, registers are automatically displayed in the CPU window. It also shows memory contents in the Memory window and the program text in the Program window. The next instruction to be executed is highlighted. The Visual Simulator has more features than the Command-line Simulator such as (virtual) peripherals, an interrupt system and a memory protection mechanism.

Load *Hello68k.asm* again in the editor. Assemble it using the "Build All" command or run the Visual Simulator directly by selecting **Visual Simulator** from the **Run** menu, typing Ctrl + F9 or by clicking the 🔍 button.

The program is shown in the Program window of the Visual Simulator and the line with the "lea"-instruction is highlighted. The program counter will indicate 00000400, this is the entrypoint of the program. Scroll through the Program window using the scrollbar on the right-hand side or with the scrollwheel of the mouse if there is one. (Scrolling with the scrollwheel is possible in Program-, Memory-, I/O - and Stack window. First click with the left mousebutton on the window to direct the scrollwheel to the desired window).

Run the program by selecting **Run** from the **Execute** menu (the Visual Simulator has its own menu), type Ctrl + F5 or click the [↓] button.

The string "Hello IDE68K!" is displayed in the I/O window which is shown automatically below the Program window. The divider can be moved to show more of the I/O window.

The I/O window is used to display or enter text. The same system calls are used as in the Command-line Simulator. However text I/O is of less importance to the Visual Simulator.

Programs like *Bounce.asm* and *Tbi68k.asm* will run on the Visual Simulator but will poorly display. This is because both programs were designed with a 80x25 text display in mind, like the one of the Command-line Simulator.

Load the program *Bsort.asm* in the editor and run it with the Visual Simulator.

Bsort.asm is a simple sorting algorithm that sorts an array of numbers in ascending order by comparing each entry with the next and exchanging both if the next entry is lower. (*Bubble sort*). Its operation is most clearly demonstrated with the simulator in Single-step mode (command **Single-step** from the **Execute** menu, function key F5 or click the [≡] button) or in Auto-step mode (command **Auto-step** from the **Execute** menu, the key combination Shift + F5 or the [≡↓] button)..

Using I/O devices.

Unlike other simulators, the Visual Simulator is equipped with virtual peripheral devices, in fact special windows that can be controlled by the 68000 program. This feature makes programming more realistic and attractive. The 68000 interacts with these devices exactly in the same way as it would interact with real hardware devices. The 68000 architecture has no special I/O instructions, but uses a part of memory which is assigned to the device. It is possible to apply not only move instructions to the device but also add, subtract, multiply, divide, clear, and, or, exclusive or, shift and rotate instructions and many others.

A peripheral device is activated by selecting the desired device in the **Peripherals** menu. It is possible to change device-parameters (such as I/O address) for each device by selecting **Configure peripherals** from the **Peripherals** menu. Reconfigure devices only when they are not active. It is important that addresses used in the 68000 program correspond with the device addresses.

Switches and LEDs.
Load the program *Switches.asm* in the editor and run it in the Visual Simulator. It is a very simple program, in fact one instruction only which copies the state of the switches to the LEDs. Click on one of the switches 0 – 7 and the corresponding LED will be turned on.

Change the program for instance to turn the LED off when the switch is turned on.

```
SWITCH   equ     $E001           ; I/O address of switches
LEDS     equ     $E003           ; I/O address of LED display

         org     $400

repeat   move.b  SWITCH,LEDS     ; read switch state and copy to LED
         bra     repeat          ; repeat
```

Figure 3. *Switches.asm*

Slider and Bar display.

These devices simulate analog I/O to the simulator. Load *Analog.asm* in the editor and run it in the Visual Simulator. The program reads the position of the slider (0 – 255) in D0 and copies it to the bar display and the LED display.

Modify the program so that an alarm sounds if the value exceeds a certain predefined limit. (see the section on the sound generator next)

```
LEDS     equ     $E003           ; I/O address of LED display
SLIDER   equ     $E005           ; I/O address of track bar
BAR      equ     $E007           ; I/O address of bar display

         org     $400

repeat   move.b  SLIDER,D0       ; read track bar position
         move.b  D0,LEDS         ; write to LED display
         move.b  D0,BAR          ; write to BAR display
         bra     repeat          ; repeat
```

 Figure 4. *Analog.asm*

Seven segment display

A seven segment display is a popular I/O device in embedded systems. It is used to display value, status and time information and diagnostic messages. The Visual Simulator has a 4-digit seven-segment display, each digit having its own I/O address. Individual bits control the segments (bit 7 the decimal point to the right of the display).

Load the program *Help.asm* in the editor and run it in the Visual Simulator. It loads a bit-pattern for HELP in the seven-segment display, waits a short period, then clears the display. Functions like this are frequently used in embedded systems to signal an abnormal situation.

Try other messages, let them rotate through the display if they are longer than 4 characters.

```
        org     $400

repeat  move.b  #%01110110,$E011     ; H  bit pattern
        move.b  #%01111001,$E013     ; E
        move.b  #%00111000,$E015     ; L
        move.b  #%01110011,$E017     ; P
        bsr     delay                ; short delay
        clr.b   $E011                ; clear display
        clr.b   $E013
        clr.b   $E015
        clr.b   $E017
        bsr     delay                ; short delay
        bra     repeat               ; repeat

delay   move.l  #$60000,D0           ; delay, depends on PC clock
loop    subq.l  #1,D0
        bne     loop                 ; loop 60000 times
        rts
```

Figure 5. *Help.asm*

Drawpad.

The drawpad device is the most complicated I/O device in the Visual Simulator. It uses six 16-bit memory locations. It is a general pixel-oriented display of variable size. Size can be set in the **Configure Peripherals** dialog from the **Peripherals** menu.

The first two words (or registers) specify a position in the display that is used as a reference point for the next operation. The coordinates must be 16-bit numbers if the size of the drawpad exceeds 256x256 pixels. The next word specifies the operation. There are codes for moving the reference position, draw a line with various colors and widths, draw rectangles (square and rounded) and ellipses (circles) of various sizes, border widths, border colors and filling colors. Another operation is the drawing of text with various colors and sizes.

The next two words can be read by the 68000 to find the position of the mousecursor in the display. For this, the mouse must be captured by clicking the left mousebutton in the drawpad display window. The mouse is released by clicking the left mousebutton with the Shift-key pressed or typing the Esc key. The last word of the device (low byte) specifies interrupt mask and IRQ level for mouse-interrupts. Interrupts may be generated for events like left- or right button pressed or released. The upper byte indicates what event caused the interrupt. It is usually processed by the mouse-interrupt handler (a 68000 function).

Load the program *Drawlines.asm* in the editor and run it in the Visual Simulator. A random pattern of lines in different colors is drawn in the pad. Change the start values of the random generator and observe that some combinations are "more random" than others.

Load the program *Drawshapes.asm* in the editor. Set the size of the drawpad device to 350x200 pixels or larger. Run the program. Text, lines and several simple figures are

drawn on the drawpad. Notice how macro's are used throughout the program to make it more readable. The macro's are defined in a separate file, *Drawpad.inc*.

Load the program *Mousedraw.asm* in the editor. With the mouse assigned to the drawpad (press left mousebutton first, cursor changes to cross-hair pattern), the user can draw on the drawpad with the mouse when the left mousebutton is down. The drawpad is erased when pressing the right mousebutton. Release the mouse by typing the Esc key or with the shift + left mousebutton combination.

The program polls the mouse-event flags in an infinite loop for left button down or right button down.

If the left button is pressed, the program enters an second loop which polls the mouse-event flags for mouse-position changed or left button up. If the mouse has moved, the program draws a line from the old position to the new position, the loop is terminated when the left mouse button is released.

If the right button is pressed, the program enters another loop which polls the event flags for right button up. When this event has occurred, the drawpad area is erased by clearing the drawpad control word and the loop is terminated. The program returns to the first loop polling the event flags for left or right mousebutton down.

Another mouse-related program, *Mouseint.asm* exercises mouse-interrupts. Flagbits are set to request an interrupt on left-button down, right-button down and mouse-moved events. The IRQ level is set to 2.

The program does not do much, the interrupt vector at $4*64 = 256$ ($100) is initialized with the address of the interrupt handler routine when the program is loaded and the interrupt bits in the 68000 status register are set to 0 (all IRQ's enabled). It then enters an infinite loop waiting for interrupts.

After an interrupt, the program checks the event-flags to see what caused the interrupt and prints a message accordingly.

Note: The mouse can only be released with the shift + left mousebutton combination, the Esc key does not work because the keyboard is assigned to the I/O window.

Sound generator.
It is possible to use the Windows® sound system as output device for the 68000. The 68000 can generate several sounds on the PC speakers by writing a special code to an I/O location. The actual sounds that are generated depend on the settings in the Configuration Menu of Windows®. Its is also possible to play a *.wav*-file on the system.

Load program *Quiz.asm* in the editor and run it in the Visual Simulator. Try other *.wav* files if the are available. The random value to guess its decimal value is taken from the low byte of the system timer. This is a 32-bit memory location at location $E040 that is incremented

at 10 millisecond intervals (100 times per second). The timer is started when the Visual Simulator starts running.

Exceptions.

Exceptions are generated by special events that require that normal program flow is interrupted and code from another function, the exception handler, must be executed. Such interrupts may be caused by internal events or by external events. One of the internal events is division by zero.

Load the program *Div2.asm* in the editor. This is a simple program that asks for two (decimal) numbers A and B, computes the result of A divided by B and displays the quotient in the I/O window. Very simple, try several numbers e.g. A = 56, B = 3 or so.

However, if B = 0 the 68000 tries to divide by zero, this is not possible and a zero-divide exception is generated. The default action of the Visual Simulator is to stop the program and give an indication of the offending instruction. Actually the instruction following the divide-instruction is highlighted because the PC already points to the next instruction when the division by zero is detected.

Often in embedded systems a zero-divide condition is caused by some malfunction of an input device. The default action, stop the program, is the last thing that we want to happen. Rewrite the program to take over the zero-divide exception and direct it to another handler that will signal the error but continues with the program, maybe with substitution of some plausible value to replace the quotient.

Interrupts.

An interrupt is an exception that is caused by some external event. Interrupts are usually generated by I/O devices. Handling an interrupt can be simulated by the Visual Simulator although in detail it differs slightly from the way the real 68000 processes an interrupt.

Interrupts usually have an Interrupt Request Level (IRQ) and an Exception vector assigned. Interrupts are only processed if the IRQ level is higher than the permitted interrupt level in the status register. The entrypoint for the interrupt handler is stored in low memory at location (4 * vector number). The 68000 has 7 so-called autovector-interrupts each corresponding to a certain IRQ level and well-defined interrupt vector. An autovector interrupt is generated by the Visual Simulator when the user clicks on one of the buttons of the toolbar I1 to I7

Load the program *Interrupt.asm* in the editor and run it in the Visual Simulator. Actually the program does not do much, it sets the interrupt vectors to the corresponding handlers and the permitted interrupt level to 4 (IRQ 5 - 7 are active). It then waits in an infinite loop for interrupts. Try the I5, I6 and I7 buttons. Interrupt 6 and 7 generate a certain sound on the PC sound system (if enabled in Windows®), interrupt 5 increments the LED display.

```
INT5       equ     $0074               level 5 autovector
INT6       equ     $0078               level 6 autovector
INT7       equ     $007C               level 7 autovector
LEDS       equ     $E003               I/O address of LEDs
SOUND      equ     $E031               I/O address of sound device


           org     $400


           move.l  #int5proc,INT5      set interrupt vectors
           move.l  #int6proc,INT6
           move.l  #int7proc,INT7
           clr.b   LEDS                turn LEDs off
           move    #$2400,SR           accept IRQ 5 – 7
wait       bra     wait                wait for interrupt

int5proc   addq.b  #1,LEDS             inc led's
           rte

int6proc   move.b  #0,SOUND            say "ping"
           rte

int7proc   move.b  #1,SOUND            say "pong"
           rte
```

Figure 6. *Interrupt.asm*

There is another invisible interrupt which is always active when the Visual Simulator is in
RUN mode. This is the hardware timer interrupt which interrupts a program at a rate of 100
interrupts per second. The timer interrupt uses vector 16 (interrupt vector address $0040)
and IRQ level 7 (non maskable). The timer is not active in Single-step or Auto-step mode.

One of the applications of this interrupt is to run a real-time operating system such as "*Mic-
rium MicroC/OS-II The Real-Time Kernel*" on the Visual Simulator. Today, multitasking
operating systems are used in many embedded systems.

Load program *Timertest.asm* in the editor and run it in the Visual Simulator. It is essentially
the same program as the previous one except that is has an additional handler for the timer.
This handler increments the bar display every 100 ms (± 10 ms) , the display bar grows
from minimum to maximum in about 256 * 100 = 25600 ms or about 25 seconds.

Supervisor mode and memory protection.
Load the program *Add5.asm* in the editor and run it with the Visual Simulator. It is a very
simple program, it adds five consecutive bytes and stores the result in the next location. Not
very spectacular.

The program is interesting for some other reason and that is not for what is does, but for
what is does not. First notice that the program runs in user-mode at memory location
$10000 and higher. A program that runs in user-mode cannot use supervisor instructions

like STOP, RESET and others, therefore the normal way of ending a program with STOP #$2700 does not work!

Also in IDE68K, memory page 0 ($0000 - $FFFF) is accessible only in supervisor mode. Consequently this program cannot make use of 68000 peripherals (unless relocated) because they reside in page 0.

Supervisor instructions, using memory in page 0 or accessing I/O devices on that page will result in a protection violation exception. Try it.

In terms op Operating System theory, page 0 and the I/O devices reside in a protected subsystem. It is only through a subroutine in supervisor space (or '*kernel*' space) that page 0 and peripherals can be accessed by user-mode programs. This subroutine or '*driver*' must be called through the TRAP mechanism[1]. A TRAP-instruction saves the current state and sets the CPU in supervisor mode. The driver routine must be terminated with a RTE instruction to restore the state of the CPU.

Starting a child process

An advanced feature of the simulators in IDE68K is the possibility to load and run another program just as it would be loaded and run with a command. The system call for this is TRAP #15 / DC.W 19.

Load the program *Kernel.asm* and run it with the Command-line Simulator or the Visual Simulator. When started, it first loads and runs the program *Div2.asm*. This program has already been loaded and run in a previous example, now it is run in user mode with initial stackpointer, status register and programcounter set by the parent.

Notice that the default handler for TRAP #15 system call has been subclassed with a special routine in *Kernel.asm*. This routine checks the word following the TRAP #15 call, if this is 0 the child process is terminated, all other values are passed to the default handler. This permits the child process to terminate with the standard exit call and still use the other system calls for printing etc.

C programming.

Understanding the inner workings of a computer and programming it in its native language (assembly) is important, however nowadays most programs are written in a high-level language such as C, C++ or Java. For embedded systems C remains the language of choice partially because of the efficiency and speed of the compiled C programs.

Unfortunately most C compilers hide the relationship between the generated machine code and the C program very well. Even if available, output from the C compiler is often in binary, hence not very well readable. Yet programmers need to have at least some under-

---

[1] See Gary Nutt, Operating Systems, 3rd ed., chap. 3, Addison Wesley, 2003

standing of what the code will look like, especially in embedded system environments where programs interact very closely with hardware such as ROM, RAM and I/O devices.

Typical textbooks on C programming never touch this subject. IDE68K fills this gap by providing a native C compiler for the 68000 family of microcomputers. Unlike most professional compilers, CC68K outputs human readable assembly code with the original C statements (optionally) inserted as comment. It is very instructive to take a look at the generated assembly code and see what a typical C statement becomes after compilation.

A C program, even after compilation to an Assembly program, is never directly executable. For instance, one of the essentials that is missing is the ORG-directive telling the assembler at what memory location the program must be loaded or execution started. A C program needs at least a small start-up program written in Assembly that contains this statement.

Also C programs frequently use functions (such as *printf, strcpy* and others) that officially are not part of the C language, but come from the standard library. Every C compiler is provided with this library. Header files (such as *stdio.h*) are also provided to assure that the functions used in the C program match the definitions in the library. This reduces the possibility of programming errors considerably.

With a program of which different parts come from different files, a *project* must be defined. A project is nothing more than a list of files that together form the program. The list may consist of any number of Assembly, C and Library files. Files of another type are allowed in a project definition but ignored when the project is processed. Properties for the project may be specified. These are options that apply to all the assembler files in the project (original assembly programs, generated assembly programs from the C compiler and modules from the library). Other options may be set for the C compiler.

Files and options are stored in a file with the suffix *.prj*. These files are regular text files and can be viewed with any standard editor (including the editor in IDE68K).

Like Assembly, programming IDE68K in C can best be illustrated by compiling and running a number of C programs from the *Examples* directory.

Traditionally, the first C program to compile and run is the program *Hello.c*. When executed, it prints the string "Hello world!" on the screen. IDE68K is no exception.

To compile *Hello.c* you must open the project *Hello.prj*. Select command **Open project** from the **Project** menu, type the F5 function key or click the [icon] button on the toolbar at the top of the main window.

Open *Hello.prj* in the dialog box.

A dialog with a list of all the files in the project is now shown. Select file *Hello.c*, click on "Open" to open the file in an edit window. Close the dialog by clicking the "OK" button. Double-clicking on the file in the list opens the file too but immediately closes the project

---

edit window.

The file *Hello.c* is now shown in the edit-window.

```
#include <stdio.h>

void main()
{
   printf("Hello world!\n");
}
```

Figure 7. *Hello.c*

To open the project dialog again, select **Edit project** from the **Project** menu, type the F6 function key or click the ![button] button in the toolbar at the top of the main window.

Open the file C*start.asm* in another edit-window.

To compile the program select **Build all** from the **Project** menu, type the F7 function key or click on the ![button] button in the toolbar.

A window is opened showing that *C:\Ide68k\Examples\Hello.c* is compiled, no errors are found, *C:\Ide68k\Lib\Cstart.asm, C:\Ide68k\Examples\Hello.asm* and C:\Ide68k\Lib\Std68k.lib are linked and *C:\Ide68k\Examples\Hello.src* is assembled.

If all is OK, no errors will be found. Warnings (if any) can be ignored at the moment.

C:\Ide68k\Examples\Hello.asm is the generated assembly file from *Hello.c*. Its contents can be loaded in an edit-window for inspection.

C:\Ide68k\Examples\Hello.src is the aggregate assembly file for the project. It is rather long and contains the modules *cstart.asm, hello.asm* and all the assembly modules for functions used in *hello.c* (like printf() called _printf in assembly). This file can also be loaded in an edit-window of IDE68K but is marked as read-only not because it is such an important file, but because editing is useless, the next compilation will overwrite all changes. The aggregate source file for a project is of little use for the programmer, but sometimes will appear when the assembler finds an error.

Finally run the program in the Command-line Simulator and the Visual Simulator and observe the result.

Introduce an error in the C program, for instance add the statement i = 5; (without a declaration for i). Compile the program with the "Build all" command or run it directly on one of the simulators.

The error will be found by the C compiler and indicated in the C program. Correct the error

and run the program again.

Introduce another error by changing the name of the printf-function to xxprintf. This function does not exist. Compile the project with the **Build all** command from the **Project** menu, F7 key or toolbar-button. Notice that it is not the C compiler or the linker that detects the error but the Assembler. Consequently the error will be indicated in *Hello.src* ("undefined symbol"). The real error however is not here but in *Hello.c*. (Another hint is the warning message "Call of function 'xxprintf' with no prototype").

Correct the error in *Hello.c* and run the program again.

Something that confuses many novice users is the fact that a project takes precedence over a file in the edit-window. As long as a project (e.g. *Hello.prj*) is active, anytime a "Run" command is executed, the program in the project will be run even if the file in the edit-window is a valid assembly program. The main window title of IDE68K indicates the active project (if any), the name of it follows the main title of "68000 IDE".

To close a project select item **Close project** from the **Project** menu or type Ctrl + F5 on the keyboard. No toolbar button is available for this command.

Try other programs like *Colors.c*. To compile and run, open project *Colors.prj*.

Colors.c is a small C program that is intended to be run on the Command-line Simulator. It shows how to use ANSI escape sequences from C. ANSI escape sequences are special strings that are not likely to be found in text and used to control color settings and cursor movements in the Command-line Simulator. ANSI escape sequences were also used in MS-DOS for this purpose. The Visual Simulator does not recognize these sequences.

Try *Sieve.c*. To compile and run, open project *Sieve.prj*. It is a C implementation of Eratosthenes' sieve to find prime numbers. You can run it on the Command-line Simulator and the Visual Simulator. It will find 1230 primes between 1 and 10000. Sieve is a benchmark for speed-testing of computers, check the performance of the IDE68K simulators against other processors.

Sieve.c is compiled with the "stack overflow detection" option on, but will run without errors. See in the assembly code how stack-checking is implemented. Checking the stack can be useful to detect nasty errors such as infinite recursive function calls etc. Note that stack-checking is performed at run-time, not at compile-time.

Sieve.c has a large global array *prime* of 10000 bytes (flags set to 1 if the index is a prime). Change the program to make this array local in the function main(). Compile and run *sieve.c* as before and observe its behavior.

Using a large array like *prime* as local variable is not practical, but certainly not forbidden. Stack overflow if fact means that the allocated amount of stack memory is exceeded. In IDE68K, the amount of memory assigned to the stack is set in a variable in *Cstart.asm*

called *stklen*. It is initially set to $1000 (or 4096). Change this value to permit enough stack-storage for *Sieve.c* to run with the local array and stack overflow detection enabled. Note that saving *Cstart.asm* with the new value will apply to all other C programs, this will not be a problem however.

Although with the local array and larger stack-storage the program runs as expected, some unexpected side-effects are introduced. To observe these effects, start the Visual Simulator with both LEDs-device and Bar-device activated. When the program runs, some activity may be observed with these devices although they are not used at all in the *Sieve.c* program.

The explanation is that the extended stack-area now overlaps with the memory region allocated to the I/O devices. While this may not bother you for the actual program, there may be circumstances where a larger stack-area or the ommision of stack-checking may result in hard-to-find runtime errors in the program.

Try other programs like *Limits.c* and *Vararg.c*. Both will run well on the Command-line Simulator and on the Visual Simulator.

Dhry.c is the internationally accepted Dhrystone benchmark program for compiler and processor testing. The program can run in the Command-line Simulator or Visual Simulator. Open project *Dhry.prj* to compile the program. Run *Dhry* on the simulator and compare the result with the results listed for the computer systems listed in the program heading.

Programming I/O devices with C.

I/O devices in a 68000 system (and in the Visual Simulator) are memory mapped, this means each register of a device corresponds to a memory location. This feature makes it possible to access each I/O device directly from within a C program by using pointers.

Also, devices with several consecutive registers in memory can be interpreted as arrays or structures. These are standard types in the C language. Using these types simplify programming with I/O devices.

The first C program that will use I/O devices is *Cswitches.c*. To compile, open project *Cswitches.prj* and run it with the Visual Simulator. *Cswitches.c* is a small program with the same functionality as the assembly program *Switches.asm.*

```
typedef unsigned char BYTE;


BYTE *switches = (BYTE *) 0xE001;
BYTE *leds = (BYTE *) 0xE003;


void main(void)
{
    for (;;)
        *leds = *switches;
}
```

Figure 8. *Cswitches.c*

Open both the compiler-generated program *Cswitches.asm* and the original assembly pro-
gram *Switches.asm* in an edit-window. It is illustrative to compare the differences between
both programs.

Also note when looking at *Cswitches.prj*, a very simple startup file called *Cswitches.a68* is
used. This file contains only one instruction, the ORG directive. Also the library file is
missing from the project list. This is possible because the program never ends and uses no
functions from the standard library. Most programs use some form of text I/O or some li-
brary function like isdigit(), strlen() however and thus need the longer *Cstart.asm* file for
startup and a library file for linking.

Run the program in the Visual Simulator, like *Switches.asm* it simply copies the state of the
switches to the LED display.

A more complicated program is *Timer.c*. To compile and run, load project *Timer.prj*.

The program displays the time in seconds since the 68000 has been started. It takes this
information from the system timer, in reality a 32-bit memory location which is incre-
mented by the hardware timer in $100^{\pm10}$ msec intervals (10 times per second). To convert to
seconds this value must be divided by 10.

Interesting is the handling of the seven-segment display. In C, this device is an array of four
16-bit short integers, *display[0]* to *display[3]*. A recursive function is used to convert the
binary value from the timer to values for the 7-segment display and to generate the bit pat-
terns for the display. For this purpose, another array, *bitpat*[] is defined, initialized with the
appropriate bit pattern codes.

Note that the program only uses functions (U)LMUL and (U)LDIV from the library. These
are functions to multiply and divide integer numbers (32 bit). If *Timer.c* would be compiled
for the 68020, which has native 32 bit multiply and divide instructions, no library function
is linked and the library file could be deleted from the projectlist.

System calls directly from C.

Another time-program is *Daytime.c*. To compile and run, load project *Daytime.prj*.

The program displays the time of day in the 7-segment display device. It takes this information from a system call in the Visual Simulator (TRAP #15 / DC.W 9) which returns the time in seconds since Jan 1, 1970, 00:00:00 hr UTC. The UNIX operating system uses the same time reference.

Date and time can be derived from this by adding the time zone difference relative to UTC first and then dividing it by proper constant values. (number of seconds per day, hour and minute). Time-functions like ctime() and asctime() in the standard library are available for most C compilers (unfortunately not for CC68K).

Interesting is the way TRAP #15 is called. Normally a small assembly-function would be required but CC68K has built-in functions _trap() and _word().

These functions can be used instead of an assembly function. Using these functions is non-portable by definition of course. Likewise pseudo-variables _D0 to _D7, _A0 to _A7 and _FP0 to _FP7 are defined. Each variable corresponds directly to the 68000 registers with that name (without underscore).

The same remark as to startup code and library usage as for the previous program applies here too.

Using a direct TRAP-call from C can be used to rewrite the program *Hello.c* in a very direct way. The program is called *Hellodir.c*. Load the project *Hellodir.prj* to compile and run it.

```
void main(void)
{
    _A0 = "Hello world!\r\n";    // A0 is pointer to string
    _trap(15);
    _word(7);                    // system call PRTSTR
}
```

Figure 9. *Hellodir.c*

Hellodir.c first assigns a pointer to the string "Hello world!" to A0 and then calls TRAP #15 function 7 (Print string).

When compiled, this program requires only 4 assembly instructions (except the startup code which can be partly omitted too).

A program like *Hellodir.c* is highly non-portable of course.

Programming interrupts in C.

The C compiler CC68K has a special keyword, *interrupt,* which, when used in a function definition, indicates to the compiler that this function is not called from within the C program but from an interrupt. This keyword applies only to programs for CC68K and is also non-portable. Functions defined as interrupt handler cannot have arguments or return a value. Local variables are permitted.

The difference between an interrupt function and a regular function is not so great, an interrupt function always saves and restores D0, D1, A0 and A1 (and other registers if they are used, like in a regular function) and returns with RTE (a regular function returns with RTS).

It is possible to call an interrupt function like a normal function. This may be useful for testing. Calling an interrupt function directly fails if the program runs in user mode.

Load the project *Cintrpt.prj* and open the file *Cintrpt.c*. The program does not do much, it sets the autovector to the interrupt function and then enters an infinite loop. The bar display is slowly incremented to indicate the program is running.

```
void *int7vec = 0x007C;        // int7 autovector
unsigned char *leds = 0xE003;  // address of LED array
unsigned char *bar = 0xE007;   // address of BAR display

interrupt void int7proc(void)
{
    (*leds)++;                 // increment LEDs at interrupt
}

void delay(void)
{
    int i = 0;

    do {
        ++i;
    } while (i < 30000);       // loop to slow things down a bit
}

void main(void)
{
    *int7vec = int7proc;       // set INT7 autovector to int func
    for (;;) {
        (*bar)++;              // increment BAR display
        delay();
    }
}
```

Figure 10. *Cintrpt.c*

The loop is interrupted when the user clicks the I7 button, control is transferred to the interrupt function where the LED display is incremented.

Running the program in Auto-step mode gives a good indication of the change in program-flow after an interrupt. (Delete or comment the call to *delay()* for this).

Putting it all together.

Load the project *Drawpad.prj* and open *Drawpad.c*. This is a C program that has everything from calling an assembly program, accessing the Drawpad I/O device (through a structure in this case) to processing interrupts in C.

The main program continuously tracks the mouse in the drawpad and displays the position on the screen. For this, the mouse must be assigned to the 68000 by first clicking the left mousebutton in the display area of the drawpad. The cursor changes to a crosshair-shaped cursor. The mouse is now uniquely assigned to the 68000 and has lost all control over Windows®. To release the mouse, click the left mousebutton with the Shift-key depressed or type Esc on the keyboard.

The mouseposition is read from the drawpad structure and converted to ASCII using the library function sprintf(). In the program, C functions are defined to draw text, lines, rectangles and ellipses on the drawpad. Other shapes are possible but not used.

When any of the mousebuttons is pressed or released, an interrupt is generated. This interrupt has IRQ 4 assigned and uses vector 64 by default.

Interrupts are enabled by setting the corresponding flags in the mouse-interrupt mask (a field in the DRAWPAD structure) and setting the interrupt level in the statusregister to 3 (IRQ 4 – 7 are accepted).

The statusregister however is not accessible from C, therefore a small assembly program must be included in the project. This program is called *Drawpad.a68* and NOT *Drawpad.asm* because this is the output from the compiler and compilation would overwrite an existing program with that name!

The assembly function is called _intlevel (called as intlevel() from C). It gets the new interrupt level from the stack, writes it in the statusregister and returns the original interrupt level. Note that only registers D0, D1 and A0 are used, these registers are free between statements and need not to be saved (unlike other registers which may contain variables when the option "use register variables" is in effect).

When the user presses or releases a mousebutton, an interrupt is generated and control is transferred to the interrupt function *buttonproc*. This interrupt procedure checks the event flags (a field in the DRAWPAD structure) to find out what caused the interrupt. If a mousebutton is pressed, the function draws a small red circle for the left- and a small green circle for the right mousebutton at a certain location in the drawpad display area. If it is found that a mousebutton is released, a small blue circle (blue is background) is drawn at the appropriate location. This effectively erases the red or green dot.

Meanwhile the hardware timer interrupts the program at approx. 100 msecs intervals. This interrupt has vector 16 assigned. The interrupt is non-maskable (IRQ 7).

A second interrupt function called *progressproc* is assigned to this interrupt. It draws a small rectangle alternatively with cyan or blue as filling color. The horizontal size of the rectangle is incremented at every interrupt and stored in a static variable. It creates a sort of progress bar at the bottom of the screen. With a maximum length of 300 pixels it will take about 2 x 300 x 100 msec = 60 sec for one cycle to complete. This time is not depending upon processor speed (but varies with processor load and Windows® activity).

Appendix: Creating a new project.

In IDE68K you can create a new project with item **New project** from the **Project** menu, there is no key or toolbar button for this function

The first file to insert is *Cstart.asm*. Click on "Insert" to insert the file. Go up one level in the directory structure and open directory *Lib*. The file *Cstart (.asm)* should be in the list. Insert the file by double-clicking on the name (or select the file and click on "Open").

Click again on "Insert" to insert another file, this is usually a C program file. Select "C programs" from the drop-down list "File types". The filename should be in the list. (If not, navigate to the directory where the file is stored). Insert the file by double-clicking on the name.

You can create a new file "on the fly" by typing the name of the file in the "Filename" window. Click on "Open", a dialog is now shown asking you to create a new file. Click "Yes" to confirm.

Repeat this procedure if you want to insert more Assembly or C files in the project.

The last file to insert is the library file *Std68k.lib*. Go up one level in the directory structure and open directory *Lib*. Select "Library files" from the drop-down list "File types". The file *Std68k (.lib)* should be in the list of files. Insert the file by double-clicking on the name.

To open a file in an edit-window, select the file you want to edit and click on "Open".

Click on "OK" if all files have been inserted.

Double-clicking on a file opens the file immediately for editing and closes the project list dialog.

Finally, the "Save project as" dialog is shown, save the new project with a non-existing name. The new project immediately becomes the active project.