Micro Digital

GoFast® x86 Floating-Point Emulators User's Guide

May 12, 2006

by Harry Ohlson

Now maintained by
Micro Digital Associates Inc.
2900 Bristol Street, #G204
Costa Mesa, CA 92626
(714) 437-7333
support@smxrtos.com
www.smxrtos.com

1 Introduction

The GoFast products help accelerate applications which execute on the Intel 80x86 architecture. This is accomplished by relinking the application with GoFast. The relinking process replaces the application's numerics with GoFast numerics.

The Intel 80x86 architecture uses a special coprocessor to execute floating-point instructions. The following table shows which coprocessor is needed in each case:

System	Processor	Coprocessor	Generation
XT, embedded	8088	8087	pre-IEEE
embedded	80188	8087	pre-IEEE
		80187	post-IEEE
XT, embedded	8086	8087	pre-IEEE
embedded	80186	8087	pre-IEEE
		80187	post-IEEE
AT, embedded	80286	80287	pre-IEEE
		80287XL	post-IEEE
AT, embedded	386	387	post-IEEE
AT, embedded	i486	none	post-IEEE

The coprocessors fall into two generations: pre-IEEE and post-IEEE. IEEE here refers to the IEEE 754 floating-point standard for microprocessors, approved in March 1985.

The pre-IEEE features are mostly a subset of the post-IEEE features. (Some pre-IEEE features were dropped, but an application would have to be really strange to even notice this.) The 80287 has a couple of control-level features not present in the 8087. The 80187 and the 80287XL are identical to the 387 except for the 32-bit mode.

The 80x86 compilers generate direct 8087 instructions in most cases. As few users actually have a coprocessor, the compilers come with an emulator. The 8087 chip is difficult to emulate exactly, and most emulators cut corners.

Computers have been getting faster and come with more memory, so math without a coprocessor is not as impractical as it used to be. Both the IEEE floating-point standard and the proposed ANSI C standard have gained wide acceptance. For these reasons, efficient and accurate emulation of the 80x87 chips has become quite important. The GoFast family of emulators and libraries was created to fill this need.

US Software designed GoFast to be:

accurate GoFast is fully compatible with both IEEE and ANSI, fully accurate, and emulates the coprocessor exactly.

fast GoFast had to be faster than anything in the market, even the corner-cutting versions.

adaptable GoFast is re-entrant, rommable and independent of DOS. It will automatically use a coprocessor if one is present, even in ROM.

usable We have produced a drop-in version for several different compilers, and will do more of these as the need arises. We also have a general version that comes with several sample installation routines.

2 Using the Drop-In Versions

2.1 GoFast / BCC

GoFast/BCC supports re-entrant floating-point calculations for the Borland C++ compiler. The following is included in library format:

floating-point emulator
_status87, _clear87, _control87, _fpreset
sqrt
sin, cos, tan
asin, acos, atan, atan2
sinh, cosh, tanh
log, log10
exp, pow
initialization (for the DOS version)

The following routines are included in source form to support linking without the Borland library:

sscanf, sprintf
floor, ceil, fabs
modf, fmod, frexp, ldexp
internal long integer math
skeleton startup routine
initialization (embedded versions)

The library routines work for all memory models; the source routines have to be compiled with the proper options. (The included makefile will do this.)

When you start using GoFast/BCC, you have two major choices to make:

1) Is the environment DOS or an embedded system? If the answer is DOS, mark your choice as "A" and skip the following question. "A" uses library USEMU.LIB which includes an automatic initialization routine.

2) Will you emulate using software interrupts or with the "coprocessor not present" interrupt? We'll call the first option B, the second C. See below for the differences, or section 4 for technical background. Both these choices use library file USEMUND.LIB, the initialization for B is in EMUINIT.ASM, for C in EMUIR7.ASM.

The following table sums up how the three choices behave:

Target	Init	Vectors	Environ	Processor	Coprocessor Use
A	AUTO	34-3e	DOS	ANY	USED if no ROM
B	PROGR	34-3e	EMBEDDED	ANY	IGNORED
C	PROGR	7	EMBEDDED	ANY EXCPT 8088/8086 V20	USED

Any of these works for all memory models. The first will not run in a non-DOS system unless the DOS environment is simulated. The second will work also in DOS, but will ignore the presence of a coprocessor. The third version uses interrupt 7: coprocessor not present. This lets GoFast take full advantage of a coprocessor when one is present, even if all the code is in ROM. The IR7 version will actually run under DOS in most cases, but there may be complications, and the computer may have to be reset afterwards.

For a simple GoFast test under DOS, just specify the library name to the linker, for instance:

```
bcc -v t1.c usemu.lib
```

or

```
tlink “path”lib\c0s t1, t1,, usemu “path”\lib\cs
```

For anything more complicated, use the provided makefile. This lets you configure the target very simply, and provides automatically the needed options.

The following table gives the timing of some floating-point operations, both with and without GoFast. The times, given in microseconds, were measured using a 16 MHz 386SX.

Operation	C++	GoFast
add	163	132
subtract	170	125
multiply	198	174
divide	205	198
sqrt	370	348
exp	1,337	1,099
log	1,154	1,081
sin	806	824
cos	788	806
tan	1,374	1,264
atan	1,264	916

These times include the C overhead, which varies from case to case, and is often substantial.

GoFast is fully rommable: it does not change any data in the code segment, it does not assume any initial value in the data segment, and it does not need DOS to run.

In the DOS version, the Borland C startup code will automatically initialize the emulator. In the embedded versions, you perform the installation with a far call to **initex87**. This can either be in the initialization routine (as in the included skeleton), or at the start of the main program.

GoFast uses interrupt 2 for unmasked exceptions. To change this, store the new number into the publicly defined byte **ex_ir**.

There are two ways to save the floating-point status at task switch:

1. Save the status with an FSAVE instruction and restore with an FRSTOR. This uses 94 bytes of space. Most multitaskers provide support for emulated FSAVE/FRSTOR. In others the user has to add the code. A few (Intel RMX) only support a coprocessor. Even this will work with GoFast if you can use the EMUIR7 initialization, that is if your processor is not 8088, 8086 or V20.

2. Use of a quick shortcut. The global word **s_stk** gives the segment of the emulator data area. The area must be 256 bytes long and **start at a new page**, in other words the segment address must end in 4 zero bits. If you allocate each task this 256-byte area, the emulator will be reentrant if you just save and restore **s_stk** at task switch time.

   Each task must call the **_fpreset** function before any other use of the emulator.

This method is somewhat of a trick; don't consider it unless you really need the speed.

The GoFast emulator uses a maximum of 110 bytes on the stack. The emulator data segments are about 400 bytes, the emulator code about 24,000 bytes.

To use USEMUIR7 in a 188/186, you must store the value of the relocation register (in hardware format) into the publicly defined location **rel_reg** before calling init87 for the first time. The power-up value is 20FF, but this is often changed in the board initialization.

GoFast/BCC uses the following segments:

where	name	class	use
initialize	_INIT_	INITDATA	code
initialize, library	_TEXT _DATA	CODE DATA	code scratch
emulator	hwseg fpcseg emuseg	FAR_BSS FAR_BSS CODE	scratch scratch code

2.2 GoFast / IC86

GoFast/IC86 provides a replacement emulator for the Intel IC86 compiler. It also includes an interface for the common C routines SQRT, SIN, COS, TAN, ATAN, ATAN2, ASIN, ACOS, LOG, LOG10, EXP and POW. There are three versions of GoFast/ IC86:

GoFast/IC86	Target System for Application Operation			
Library Name	**Vectors**	**Environ**	**Processor**	**Coprocessor Use**
USEMU.LIB	D4-DF	DOS/EMBD	ANY	IGNORED
USEMUND.LIB	14-1F	EMBEDDED	ANY	IGNORED
USEMUIR7.LIB	7	EMBEDDED	ANY EXCPT 8088/8086 NEC V	USED

Any of these replaces the Intel libraries DE8087.LIB and DE8087 for all memory models. The first two differ only in the interrupt vectors they use. The third version uses interrupt 7: coprocessor not present. This lets GoFast take full advantage of a coprocessor when one is present, even if all the code is in ROM. The IR7 version will actually run under DOS in most cases, but there may be complications, and the computer may have to be reset afterwards.

To speed up your application simply relink it using the GoFast emulator and library. You can also link in just the emulator; this alone will give you a big speed improvement.

To replace both the Intel emulator and the common Intel routines with GoFast, link as follows:

```
link86 cstartL.obj, MYPROG.obj, &
    usemu.lib, &
    clibL.lib, cfloatL.lib, cdosL.lib, cel87.lib &
to MYPROG.exe exe
```

If you only want to replace the Intel emulator, link as follows:

```
link86 cstartL.obj, MYPROG.obj, &
    clibL.lib, cfloatL.lib, cdosL.lib, cel87.lib, &
    usemu.lib &
to MYPROG.exe exe
```

The following table gives the timing of some floating-point operations, both with and without GoFast. The times, given in microseconds, were measured using a 16 MHz 386SX.

Operation	IC86	GoFast
add	745	150
subtract	765	152
multiply	862	194
divide	1,540	214
sqrt	5,917	223
exp	11,203	987
log	8,733	933
sin	11,187	697
cos	11,827	714
tan	8,127	1,117
atan	8,420	770

These times include the C overhead, which varies from case to case, and is often substantial.

GoFast is reentrant in the same way as the floating point chip. As with the coprocessor, a task switch is done by using an FSAVE to save floating point status, and by using an FRSTOR to restore floating point status.

GoFast is fully rommable: it does not change any data in the code segment, it does not assume any initial value in the data segment, and it does not need DOS to run.

The C startup code will automatically initialize the emulator. In cases where this is not done, you can perform the installation with a far call to **init87.**

GoFast uses interrupt 2 for unmasked exceptions. To change this, store the new number into the publicly defined byte **ex_ir**.

The GoFast emulator uses a maximum of 110 bytes on the stack.

To use USEMUIR7 in a 188/186, you must store the value of the relocation register (in hardware format) into the publicly defined location **rel_reg** before calling init87 for the first time. The power up value is 20FF, but this is often changed in the board initialization.

GoFast/IC86 uses the following segments:

where	name	class	use
initialize, library	CODE	CODE	code
emulator	hwseg	BSS	scratch
	xseg	BSS	scratch
	emuseg	CODE	code

2.3 GoFast / MSC

GoFast/MSC supports re-entrant floating-point calculations for the Microsoft C compiler versions 5.1, 6 and 7. The following is included in library format:

floating-point emulator
_status87, _clear87, _control87, fpreset
sqrt
sin, cos, tan
asin, acos, atan, atan2
sinh, cosh, tanh
log, log10
exp, pow
initialization (for the DOS version)

The following routines are included in source form to support linking without the Microsoft library:

sscanf, sprintf
floor, ceil, fabs
modf, fmod, frexp, ldexp
hypot, cabs
internal long integer math
skeleton startup routine
initialization (embedded versions)

The library routines work for all memory models; the source routines have to be compiled with the proper options. (The included makefile will do this.)

When you start using GoFast/MSC, you have two major choices to make:

1. Is the environment DOS or an embedded system? If the answer is DOS, mark your choice as "A" and skip the following question. "A" uses library USEMU.LIB which includes an automatic initialization routine.

2. Will you emulate using software interrupts or with the "coprocessor not present" interrupt? We'll call the first option B, the second C. See below for the differences, or section 4 for technical background. Both these choices use library file USEMUND.LIB, the initialization for B is in EMUINIT.ASM, for C in EMUIR7.ASM.

The following table sums up how the three choices behave:

Target	**Init**	**Vectors**	**Environ**	**Processor**	**Coprocessor Use**
A	AUTO	34-3e	DOS	ANY	USED if no ROM
B	PROGR	34-3e	EMBEDDED	ANY	IGNORED
C	PROGR	7	EMBEDDED	ANY EXCPT 8088/8086 V20	USED

Any of these works for all memory models. The first will not run in a non-DOS system unless the DOS environment is simulated. The second will work also in DOS, but will ignore the presence of coprocessor. The third version uses interrupt 7: coprocessor not present. This lets GoFast take full advantage of a coprocessor when one is present, even if all the code is in ROM. The IR7 version will actually run under DOS in most cases, but there may be complications, and the computer may have to be reset afterwards.

For a simple GoFast test under DOS, specify the library name to the linker using the NOE option, for instance:

```
cl –c t1.c

link /NOE t1,,,usemu;
```

NOE is needed to tell the Microsoft linker that the same symbols are defined in two different libraries. For anything more complicated, use the provided makefile. This lets you configure the target very simply, and provides automatically the needed options.

If you forget the NOE option, the link will either fail or produce an unusable load file. (But if you use NOD for "no standard libraries", the NOE is not needed.)

Compiler option Aw is usually needed in multitasking compact and large models. (Leaving it out will in practice hurt floating-point the most.)

The following table gives the timing of some floating-point operations, both with and without GoFast. The times, given in microseconds, were measured using a 16 MHz 386SX.

Operation	MSC	GoFast
add	165	128
subtract	172	128
multiply	220	174
divide	247	194
sqrt	608	271
exp	2,527	990
log	2,067	987
sin	2,840	697
cos	2,800	713
tan	2,197	1,137
atan	2,217	787

These times include the C overhead, which varies from case to case, and is often substantial.

GoFast is fully rommable: it does not change any data in the code segment, it does not assume any initial value in the data segment, and it does not need DOS to run.

In the DOS version, the Microsoft C startup code will automatically initialize the emulator. In the embedded versions, you perform the installation with a far call to **initex87**. This can either be in the initialization routine (as in the including skeleton), or at the start of the main program.

GoFast uses interrupt 2 for unmasked exceptions. To change this, store the new number into the publicly defined byte **ex_ir**.

There are two ways to save the floating-point status at task switch:

1. Save the status with an FSAVE instruction and restore with and FRSTOR. This uses 94 bytes of space. Most multitaskers provide support for emulated FSAVE/FRSTOR. In others the user has to add the code. A few (Intel RMX) only support a coprocessor. Even this will work with GoFast if you can use the EMUIR7 initialization, that is if your processor is not 8088, 8086, or V20.

2. Use a quick shortcut. The global word **s_stk** gives the segment of the emulator data area. The area must be 256 bytes long and **start at a new page**, in other words the segment address must end in 4 zero bits. If you allocate each task this 256-byte area, the emulator will be reentrant if you just save and restore **s_stk** at task switch time.

   Each task must call the **_fpreset** function before any other use of the emulator.

   This method is somewhat of a trick; don't consider it unless you really need the speed.

Unfortunately the Microsoft C compiler stores the return value from type **double** functions into the global variable _fac. This is **not reentrant**, not even when a coprocessor is used. The only satisfactory solution is to have the real-time executive save and restore _fac. This is 8 bytes in Microsoft release 5.1, 10 bytes in releases 6 and 7.

If you can't get the scheduler to save _fac, you can perhaps still use the compiler in multitasking with some limitations:

- Compiler option / Oi takes out the _fac use from all intrinsic functions (SQRT, SIN, etc.).

- Function type **long double** does not use _fac, and is not necessarily any slower than **double**.

The GoFast emulator uses a maximum of 110 bytes on the stack. The emulator data segments are about 400 bytes, the emulator code about 24,000 bytes.

To use USEMUIR7 in a 188/186, you must store the value of the relocation register (in hardware format) into the publicly defined location **rel_reg** before calling init87 for the first time. The power-up value is 20FF, but this is often changed in the board initialization.

GoFast/MSC uses the following segments:

where	name	class	use
initialize	CDATA	DATA	code
initialize, library	_TEXT	CODE	code
	_DATA	DATA	scratch
emulator	hwseg	FAR_BSS	scratch
	fpcseg	FAR_BSS	scratch
	emuseg	CODE	code

2.4 GoFast / HIGHC

GoFast/HIGHC provides a replacement emulator for the Metaware High C 386/486 compiler version 3. The following is included in library format:

floating-point emulator
initialization

The following routines are included in source form:

floor, ceil
atan2
sinh, cosh, tanh
pow
hypot
sscanf, sprintf
modf, fmod, frexp, ldexp
conversion from double to integer
initialization

(Some common math routines are missing from the list. This is because the compiler will generate in-line code for them, and there is no convenient way to use a subroutine.)

You’ll find the file names and the needed compilation and link commands in the included makefile.

The emulator and the installation routine are in a Pharlap format library. There are two versions: USEMU.LIB for flat addressing, and USEMUSEG.LIB for segmented addressing. (The standard versions of the High C library and the Pharlap linker only support flat addressing, but segmented versions for embedded systems exist.)

GoFast/HIGHC is a true emulator: it uses the “coprocessor not present” interrupt. The emulator will use a 387 if one is present, unless the DOS environment variable “NO87” is set.

Because GoFast is a true emulator, you have to compile the source with the 387 option. To link in GoFast, include the GoFast library and the system libraries. An example might look like this:

```
hc386 –f387 –priv test1.c
```

(The “priv” option is needed so that GoFast can install the coprocessor interrupt under the Pharlap DOS extender.)

The GoFast installation routine is for the Pharlap DOS extender. Installing interrupts in protected mode can be tricky, and the provided method might not work under some other extender or under a multitasking executive. We are including the source for a couple of different installation routines.

The following table gives the timing of some floating-point operations, both with and without GoFast. The times, given in microseconds, were measured using a 16 MHz 386SX.

Operation	HIGHC	GoFast
add	110	117
subtract	143	117
multiply	124	135
divide	366	146
sqrt	1675	145
exp	2653	790
log	3550	493
sin	2417	350
cos	2710	367
tan	3040	567
atan	3060	420

These times include the C overhead, which varies from case to case, and is often substantial.

Metaware uses library calls for floating-point emulation. A native High C program compiled with the 387 option will not run without a coprocessor. A native High C program compiled with the default options will run with or without a coprocessor, but does not use the 387 very efficiently. Just linking in GoFast will remove these limitations.

GoFast is reentrant in the same way as the floating point chip. As with the coprocessor, a task switch is done by using an FSAVE to save floating point status, and by using an FRSTOR to restore it.

GoFast is fully rommable: it does not change any data in the code segment, it does not assume any initial value in the data segment, and it does not need DOS to run.

The C startup code will automatically initialize the emulator. In cases where this is not done, you can perform the installation with a call to **inite387**.

GoFast uses interrupt 2 for unmasked exceptions. To change this, change the variable EXIRNO in the initialization code and replace the initialization module in the library. (Some hardware uses interrupt 16.)

As a default, GoFast enables interrupts right at the entry if they were enabled at the main level. You can turn off this feature by changing the variable ENABLE in the initialization module to 0. (In some environments enabling of interrupts is either very slow or prohibited.)

The GoFast 32-bit emulator uses a maximum of 160 bytes on the stack.

GoFast/HIGHC uses the following segments:

where	name	class	use
initialize	_TEXT	CODE	code
emulator	hwseg	DATA	scratch
	emuseg	CODE	code

2.5 GoFast / WCC

GoFast/WCC provides a replacement emulator for the WATCOM C 386/486 compiler.

The emulator and the installation routine are in a WATCOM format library. There are two versions: USEMU.LIB for flat addressing, and USEMUSEG.LIB for segmented addressing. (The standard versions of the WATCOM library and linker only support flat addressing, but segmented versions for embedded systems exist, see WATCOM for details.)

GoFast/WCC is a true emulator: it uses the "coprocessor not present" interrupt. The emulator will use a 387 if one is present, unless the DOS environment variable "NO87" is set.

To link in GoFast, you just need to link in the library:

```
wc1386 prog.c usemu.lib
```

The GoFast installation routine is for the Pharlap DOS extender. Installing interrupts in protected mode can be tricky, and the provided method might not work under some other extender or under a multitasking executive. We are including the source for a couple of different installation routines; see README.TXT for details.

The following table gives the timing of some floating-point operations, both with and without GoFast. The times, given in microseconds, were measured using a 16 MHz 386SX.

Operation	WCC	GoFast
add	238	132
subtract	244	135
multiply	242	148
divide	265	156
sqrt	361	295
exp	1683	1447
log	900	770
sin	880	477
cos	857	457
tan	1487	657
atan	1117	493

These times include the C overhead, which varies from case to case, and is often substantial.

GoFast is reentrant in the same way as the floating-point chip. As with the coprocessor, a task switch is done by using an FSAVE to save floating point status, and by using an FRSTOR to restore it.

GoFast is fully rommable: it does not change any data in the code segment, it does not assume any initial value in the data segment, and it does not need DOS to run.

The modified C startup code will automatically initialize the emulator. In cases where this is not done, you can perform the installation with a call to **inite387**.

GoFast uses interrupt 2 for unmasked exceptions. To change this, store the new number into the publicly defined byte **ex_ir**.

The GoFast 32-bit emulator uses a maximum of 160 bytes on the stack.

GoFast/WCC uses the following segments:

where	name	class	use
initialize	_TEXT	CODE	code
emulator	hwseg	DATA	scratch
	emuseg	CODE	code

2.6 GoFast / AT

GoFast/AT is a free-standing floating-point emulator for the 80188, 80186, 80286, 386 and i486SX processors. It is installed as a resident program (TSR), not linked with the application. GoFast/AT fools the application into believing there is a coprocessor, and will therefore work even for programs that have been written to require a coprocessor.

There are two versions of GoFast/AT. The 16-bit version EMUL16 will work in all the processors listed above. The 32-bit version EMUL32 is faster, but requires either a 386 or an i486. GoFast/AT will run under MS/DOS in real mode or in virtual 8086 mode.

To install the emulator, just run the corresponding program:

emul16	install 16-bit version
emul32	install 32-bit version
emul16 off **emul32 off**	remove any old version

If either of these is already installed, the installation command just removes the old version, and you’ll have to repeat it. (We don’t automatically re-install the emulator, because this would make freeing the memory quite difficult.)

The following table gives the timing of some floating-point operations, first for the Microsoft emulator, and then for GoFast/AT16 and GoFast/AT32. The times, given in microseconds, were measured using a 16 MHz 386SX.

Operation	MSC	AT16	AT32
add	165	135	124
subtract	172	143	123
multiply	220	181	143
divide	247	205	150
sqrt	608	529	454
exp	2,527	2,763	1,977
log	2,067	1,337	897
sin	2,840	1,100	790
cos	2,800	1,117	787
tan	2,197	1,977	1,353
atan	2,217	1,317	970

These times include the C overhead, which varies from case to case, and is often substantial. The times were derived using standard Microsoft libraries.

GoFast/AT is re-entrant in the same way as the floating-point chip: FSAVE saves the status of the emulator, FRSTOR restores it. GoFast/AT is not meant for embedded systems.

GoFast/AT gets control through interrupt 7: coprocessor not present. Unfortunately there are programs that use this interrupt vector improperly; one of these is the Microsoft CodeView. Trying to CodeView a program that needs the interrupt 7 vector results in an immediate crash. Please note that the Borland development tools have no such difficulties.

The GoFast installation code has to enable interrupt 7. This can cause problems if you are using protected mode: a DOS extender or a memory manager. Some of these, such as the Quarterdeck QEMM, recognize changing the emulation mode as legitimate, others, such as the Microsoft Windows, do not. Future changes to the DOS extender standard will probably solve this problem.

One of the biggest problems in DOS extenders is what they can do to interrupt overhead. In some extreme cases any kind of floating-point emulation becomes basically impossible. Fortunately some memory managers (QEMM for one) are not nearly this bad.

There are two common methods for determining if a coprocessor is present. Some programs just try out floating-point instructions to see if they work. Because GoFast has set the automatic emulation mode, the code will work exactly as if a chip were present. Some programs use the BIOS configuration query INT 11. GoFast intercepts this call and turns on the coprocessor bit.

Using GoFast/AT in a 188/186 requires the relocation register to be configured for interrupt 7. This should be done either by the ROM initialization, or by a simple startup program. GoFast has no way of finding out where the relocation register is. There are very few PC’s built around an 188/186.

Even though GoFast/AT32 uses 32-bit registers, it is not a true 32-bit emulator. There is not practical way to support all the different 80x86 modes in one emulator.

3 Using the General Versions

3.1 GoFast/EMU

GoFast/EMU is a floating-point emulator for the Intel 80x86 processors. It also includes an interface for the elementary functions SQRT, SIN, COS, TAN, ATAN, ATAN2, ASIN, ACOS, LOG, LOG10, EXP and POW.

GoFast/EMU supports **real** and **virtual 8086** mode only: it will not run in protected mode in 286/386/486.

GoFast/EMU is intended for those situations which the drop-in versions do not cover. It comes with several different installation routines in source format; one of them may or may not fit your particular need.

The installation routine will install the calling interface either to the emulator, or to code that uses a coprocessor. Whether a coprocessor is used depends of course on whether one is present, and possibly also on an environment variable, such as “NO87=1”. An embedded application will usually not worry about the configuration; this will after all be pretty much fixed for good.

For software emulation, you need to install one or more of the following interrupt handlers:

emu87	emulation using interrupt 7
emuwait	fwait interrupt
emua	emulation with software interrupts, Microsoft 34-3B or Intel D8-DF
emub	segment override interrupts, Intel 34-37, Microsoft/Borland 3C
emuc	library functions for Borland interrupt 3E

To use a coprocessor with software interrupts (not with interrupt 7), you need to install interrupt handlers that take advantage of it. GoFast comes with sample versions for these:

emuwah	fwait interrupt
emuah	emulation with software interrupts, Microsoft 34-3B or Intel D8-DF
emubh	segment override interrupts, Intel 34-37, Microsoft/Borland 3C
emuch	library functions for Borland interrupt 3E

Interrupt 7 emulation requires the installation of the emu87 routine, also emuc/emuch if Borland/Microsoft library functions are used.

After the interrupt vectors have been installed, the emulator should be initialized with an FNINIT instruction.

GoFast/EMU contains the emulator in object format in two versions, one for the Intel linker, and one for the other linkers. All other files are in source format; see the included README.TXT file for inventory.

You have to consider the included function interfaces as samples only, because no commonly accepted argument interface exists. Also, they contain code for ANSI C error handling.

The sample installation code sets up interrupt 2 as the unmasked exception; this is what most PC’s use. You can change the value by changing EXIRNO in the parameter file EMU.INC.

The GoFast emulator uses a maximum of 110 bytes on the stack.

To use USEMUIR7 in a 188/186, you must store the value of the relocation register into the publicly defined location **rel_reg** before calling the initialization for the first time. The power-up value is 20FF, but this is often changed in the board initialization.

The emulator itself uses one code segment (name emuseg, class CODE) and two scratch segments (names hwseg and xseg, class BSS).

3.2 GoFast/PROT

GoFast/PROT is a floating-point emulator for the 386 and i486SX 32-bit protected mode. It is distributed in source format, to support a wide variety of environments.

GoFast/AT is re-entrant in the same way as the floating-point chip: FSAVE saves the status of the emulator, FRSTOR restores it.

GoFast is fully rommable: it does not change any data in the code segment, it does not assume any initial value in the data segment, and it does not need DOS to run.

GoFast/PROT gets control through interrupt 7: coprocessor not present. The GoFast installation code generally has to perform the following functions:

1. Check if there is a coprocessor, skip steps 2 and 3 if yes
2. Install an interrupt descriptor for interrupt 7.
3. Enable the emulation interrupt.
4. Initialize the emulator or the coprocessor.

GoFast/PROT contains a make file, the emulator source, and a sample initialization module. There are two versions of the main level: one for the flat model and one for the segmented model. The source was written for the Pharlap assembler; using some other 386 assembler will probably present few if any problems. See the included README.TXT for an inventory.

The sample installation code sets up interrupt 2 as the unmasked exception; this is what most PC's use. You can change the value by changing EXIRNO in the parameter file EMU.INC.

The GoFast 32-bit emulator uses a maximum of 160 bytes on the stack.

The emulator uses one code segment (name emuseg, class CODE), and one scratch segment (name hwseg, class BSS).

4 Technical Background

4.1 Emulation Interface

Several different emulation interfaces are in use. These are the most common:

1. Emulation with **hardware**. The Intel 80186, 80188, 80286, 386 and i486SX processors can be configured so that a floating-point instruction causes an interrupt 7 when there is no coprocessor. The NEC V33/V53 causes an interrupt 130 when there is no coprocessor.
2. The **Intel IC86** emulation. Interrupts D8-DF are used for the ESCO-ESC7 instructions. Interrupts D4-D7 signify segment overrides.
3. The **Intel PL/M** emulation. Interrupts 18-1F are used for the ESCO-ESC7 instructions, 14-17 for segment overrides. This will not work in a PC
4. The **Microsoft** emulation. Interrupts 34-3B are used for ESCO-ESC7. Interrupt 3C is used for segment overrides, and 3D for the fwait instruction. The Borland variation also uses 3E for library functions.

In the first of these, the emulated code consists of floating-point instructions as such, with nothing added. In the others, the compiler has to add an extra byte to each instruction. At link time, the instruction is fixed to be either an interrupt or an actual floating-point instruction.

We'll give an example on how the fixing works. We start from the instruction

```
FPREM = D9 F8
```

The compiler actually generates

```
extrn       FPFIX: abs
dw          FPFIX+d990h
db          0F8h
```

Now, if FPFIX has the value 003D, the sequence will become **CD D9 F8**, that is an INT D9 followed by the second instruction byte. If FPFIX is given the value 0, we get **90 D9 F8**, that is a no-operation followed by an FPREM. The exact details vary, but the idea is to let the same object code serve both an emulator and a coprocessor.

Often it is not enough that the same object code can be linked either for emulation or for a coprocessor. There are two different ways of supporting completely dynamic reconfiguring, where the same load file can run with or without a coprocessor. In the first of these, the program originally uses software interrupts for floating-point instructions. If there is a coprocessor, the interrupt routine will alter the interrupted instruction to a real coprocessor instruction, and re-execute it. The resulting code is fairly efficient (at least if there are plenty of loops), but will definitely not work in a ROM.

The second method depends on the “coprocessor not present” interrupt, which is more efficient and will work even in ROM. However, it can’t be used in the Intel 8088/8086, or the NEC V series except for V33/V53.

GoFast offers both of the above options; Microsoft and Borland only the first; Intel IC86 neither.

4.2 Floating-Point Exceptions

The requirements for floating-point exceptions are really the same as for any exceptions. Errors should be detected, and treated in a reasonable way. Let's start from a simple example:

```
int i1, i2, i3;
i3 = (100 * i1) / i2;
```

This piece of integer arithmetic has two exceptions: divide by zero and overflow. In a typical case, dividing by zero crashes the program but never happens, overflow is ignored and even intentional sometimes.

Exceptions for the floating-point counterpart

```
float f1, f2, f3;
f3 = (100. * f1) / f2;
```

should, if anything, be less of a concern, because overflows don't happen easily in floating-point numbers. You should be very much concerned about detecting the errors, but not about sophisticated treatment of them, such as can be found in the IEEE standard. Any system that pushes through invalid data is still in testing; the objective should be to fix the code, not to try to get right results from wrong data.

The IEEE 754 standard contains three basic mechanisms for detecting errors. **Masked exceptions** set a sticky bit in a status word, and you can at your discretion test the bits. A masked exception causes a **special value** to be returned from the operation. **Unmasked exceptions** cause an immediate interrupt. There are six different exception types; each can be configured as masked or unmasked.

ANSI C is, unlike IEEE 754, a very flexible standard. All kinds of error handling schemes are in use; most neither defined nor prohibited in the standard. These are the most common:

1. The IEEE 754 exception handling is available in most 80x86 C compilers. The drawback is that the code may not be portable, also some C libraries are written so that IEEE exceptions are not always raised

2. ANSI C requires that the intrinsic math routines (SQRT, SIN etc.) return an error code **errno**. Unfortunately errors from user code are not detected this way. For instance: f1 = pow(0, -1) causes an error, f1 = 1/0 does not.

3. Most C compilers support use of the **matherr** routine. This is the unmasked counterpart of the **errno** system: matherr gets called whenever errno is set. The method has therefore all the drawbacks of the errno method, it also tends to cause problems in embedded systems.

   Matherr is **not** standard ANSI C.

Exception handling in C tends to suffer from the following problems:

- not all errors are detected
- code becomes non-portable
- error handling is not re-entrant
- program needs DOS to run

GoFast can't solve all these problems, but it helps because it is fully IEEE-compatible. The simplest scheme might be something like this:

- Unmask the serious exceptions, that is invalid operation, zero divide, overflow. (In some tightly controlled applications, you could unmask even underflow and denormal.)
- Install routine "catastrophic error", whatever that means in your case, as interrupt 2 handler.
- Weed out improper data in application code, so that there will be no improper operations

4.3 Accuracy in Floating-Point Calculations

Floating-point calculations are in practice always inexact. This is easy to forget because just about everything else in programming is exact, and because the precision seldom becomes a problem. But you forget only at your own peril.

There is nothing mysterious about the loss of precision, it’s simply the nature of the thing. The rest of this chapter illustrates different faces of the inaccuracy.

Rounding

A floating-point number contains a fixed number of digits. Unless there are a lot of trailing zeroes, an arithmetic operation will very likely produce too many digits to fit in the same space. This of course happens even in normal decimal calculations, for instance:

```
  1234.567
+    12.34567
  1246.91267 => 1246.913
```

Rounding errors as such are unlikely to become noticeable, but they can be enhanced by other effects. Some algorithms are notoriously prone to lose precision.

Base Conversion

Changing the base of a fractional number generally requires approximations. Any application that uses decimal input, decimal constants or decimal output has to perform base conversions. Consider the example

```
float f1;
f1 = 1.1;
printf (“%.12; f\n”, f1);
```

This program will display the value 1.100000023842, not the exact 1.1. What happened?

The root of the problem is that 1 1/10 in base 2 is 1.0001100 (1100), i.e. can’t be represented exactly. The compiler creates a constant 1.1 with 24 bits:

```
1.000 1100 1100 1100 1100 1101
```

This value is obviously larger that 1.1 because we rounded up at bit 24. Printing the value with too many decimals (anything more that 7 in this case) will show the difference.

Difference Between Large Numbers

Let’s try the program

```
float f1, f2, f3;
f1 =1234.0;
f2 = 1233.1;
f3 = f1 - f2;
```

```
printf(“%1f\n”,f3);
```

The result is 0.900024, i.e. off by quite a bit. The basic effect is the same as explained above: the required base conversion. But the relative error got enlarged in the subtraction of two almost equal numbers:

```
  1234.0 = 1001 1010 010.0 0000 0000 0000
- 1233.1 = 1001 1010 001.0 0011 0011 0011
                             0.1 1100 1100 1101
```

Irrational Numbers

Values such as sqrt(2) or sin(0.5) have no exact representation in any base. These can still be calculated “exactly” to the value that is mathematically correct considering the rounding rules. IEEE specifically requires an exact square-root, but says nothing about other functions. The GoFast square-root is of course exact.

You probably won’t find an “exact” implementation of the transcendentals anywhere. The additional error should be of the same order as the rounding error. The Intel coprocessor (and consequently GoFast) calculates the transcendentals to 64 bits of precision, and rounding this to **float** or **double** will of course produce an exactly correct result most of the time.

Special Functions

As a rule, the relative error of a function is different than the relative error of the argument. In some cases this becomes important. Take the following code:

```
double d1, d2;
d1 = 1.1;
d3 = exp(100 * d1);
```

The result will differ from exp(110) by quite a bit. This does not mean that exp(x) is inaccurate; it means that the original inaccuracy of x got magnified. An important special case is

```
z = x ^ y = e ^ (y * log(x))
```

This formula is **not** a satisfactory way of calculating pow(x,y). (GoFast of course uses better methods.)

A point where a function approaches zero for a non-zero argument is especially tricky. As an example, log(0.999998) is close to twice log(0.999999). If your argument is only a little inexact, say due to rounding, the answer may be so wrong as to be meaningless. Again we need to remember that log(x) as such is not the culprit, it is not inaccurate.

The same warning applies whenever significant argument reduction is needed, such as the trigonometric functions for arguments much larger than pi. Worst of all are cases where these two situations coincide: sin(1000 * pi) for instance.

Conversion to Integer

ANSI C specifies that a floating-point number is converted to an integer using **truncation**: the decimals are discarded. This innocuous rule can cause surprises. Consider the program

```
int i1, i2;
i1 = 256;
i2 = (float) i1/2.56;
printf("%d\n",i2);
```

Certainly the correct answer is 100, but you **can't** count on this; the program as written is unstable. On many systems, the answer will keep jumping between 99 and 100, depending on the compilation options and the exact code used.

The root reason for the instability is not hard to see. The value 2.56 has to be rounded when it is converted to base 2. If this rounding is **up**, the division will give a value that is slightly **less** than 100. According to ANSI C rules, this becomes 99. If again 2.56 in base 2 is rounded **down**, the division will give slightly **over** 100, and truncates to 100.

IEEE 754 is a very rigorous standard, and whether 2.56 is rounded up or down, surely it should be rounded the same way every time. How is it possible that two standard implementations give different results?

Well, it really isn't. This is an interesting example of what happens when a standard meets reality. Let's look at the rounding first.

How the rounding is done depends on the number of bits in the constant. ANSI C says that a floating-point constant is **double**, and IEEE 754 rules this to have 53 binary digits. Unfortunately:

1. Some compilers use **float** constants in **float** expressions. This difference may be enough to change the direction of the actual rounding
2. Some compilers optimize out all divisions by a constant, using instead a multiplication with the inverse value. What happens to the rounding is anybody's guess.

In the 80x86, the most common reason for these discrepancies is the fact that the coprocessor uses 64-bit internal precision. Here are two possible ways to compile the above program:

1.
```
fild   i1          ; i1 to internal FP format
fdiv   2.56        ; divide FP stack by 2.56
fstp   w1          ; resulting float to temp
fld    w1          ; result again to FP stack
call   toint       ; simple routine to truncate
fistp  i2          ; final result
```

```
2.  fild   i1          ; i1 to internal FP format
    fdiv   2.56        ; divide FP stack by 2.56
    call toint         ; simple routine to truncate
    fistp i2           ; final result
```

These two sequences are not equivalent, because the internal floating-point registers hold 64 bits, and the temporary variable only 24. Moving the internal register into a single-precision variable requires rounding. This could quite possibly bring the value slightly over, or slightly under, an integer value.

Is it legal to change floating-point precision internally? Nobody seems to really know. (The Intel coprocessors have a constant-precision mode, but nobody is using this.) The answer may not matter that much really, because ANSI C anyway allows for variation in the order in which operations in an expression are performed. At the level where rounding becomes important, the value of an expression generally depends on the exact order of calculations.

5 References

ANSI/IEEE Standard 754-1985: Binary Floating-Point Arithmetic

ANSI Document X3J11/88-159: Draft Proposed American National Standard for Information Systems – Programming Language C

387 DX User's Manual, Intel Corporation #231917-002, 1989

W. Cody, W. Waite: Software Manual for the Elementary Functions, Prentice-Hall, 1980