Professional Pascal 2.7 and High C 1.4 Upgrade Information

The items below are new relative to the initial release of Professional Pascal 2.6 and High C 1.3. Items apply to both Pascal and C unless otherwise indicated. The items are documented either in the on-line README file, or in the **Programmer's Guide** (PG) as indicated.

Where Doc'd	Subject
README	In-line code — the ability to place constants directly into the instruction stream. (MS-DOS only)
README	`-make` command-line option assists in the construction of make files.
README	For embedded applications, the supplied utility `bd` can analyze object files and libraries for occurrences of initialized data. (Intel OMF targets only)
README	`#include` can be made to behave in a non-relative fashion.
README & PG	Toggle `387` allows generation of in-line 80387 code, which makes excellent use of the 387's in-line transcendental instructions.
README	DOS 3.x networking is supported.
PG	For the real-mode 8086/186/286/386 compilers, toggle `386` allows generation of real-mode 386 code. All new instructions are used except for those using 32-bit registers. (80386 targets only)
PG	The run-time library senses the presence of a 386 and does long divides using the 386 native instructions. (MS-DOS only)

PG — More optimizations have been added. Some are controlled by the following toggles:

Toggle `Optimize_FP` (Intel targets only)
Toggle `Mpy_8086` (Intel targets only)
Toggle `Push_regsize` (Intel targets only)

README — Far pointers for data (Professional Pascal) and Near/Far pointers for data and code (High C) are supported. (Intel targets only)

PG — New calling convention attributes:

Professional Pascal
`Return_aggregate_as_pointer`
(Intel targets only:)
`Return_32_in_bx_ax`
`Near_call`
`Far_call`

High C:
`_RETURN_AGGREGATE_AS_POINTER`
(Intel targets only:)
`_RETURN_32_IN_BX_AX`
`_NEAR_CALL`
`_FAR_CALL`

The last two attributes in each case allow non-standard routine/function linkage.

PG — Additional documentation and library support have been provided for reducing the size of an executable file. (Intel targets only)

PG — 80-bit floating-point numbers (`ExtReal` in Professional Pascal, **`long double`** in High C) can now be both input and output.

PG — Miscellaneous (Intel targets only):

Toggle `Emit_empty_groups`
Toggle `Group_data_externs`
Toggle `Group_code_externs`
Pragma `Dclass`
Pragma `Cclass`

High C only:

PG	Function prototypes can now be mixed with old-style function definitions. In addition, the compiler warns whenever the semantics of an old-style definition is overridden by a prototype. The warning is toggle-controllable.
README	ANSI-specified **const** and **volatile** are implemented.
README	__HIGHC__ is defined to be 1 in all implementations of High C.
PG	Signed bit fields are supported (as required by the ANSI standard). (Intel targets only)
LRM	Structure members can be aligned or not via the keywords **_packed** and **_unpacked**. See the **Language Reference Manual** (LRM).
LRM	The ANSI-mandated `offsetof` macro is supported via the intrinsic `_offsetof` function. See the **Language Reference Manual.**
LbRM	The `system` function is implemented for executing a subprocess. See the **Library Reference Manual** (LbRM).
PG	Miscellaneous: Toggle `Struct_by_value_warnings` Toggle `Prototype_conversion_warn` Toggle `Prototype_override_warnings` Toggle `Char_default_unsigned`

MetaWare High C Release 1.3 New Features

This document describes what has been added to release 1.3 of High C over release 1.2a. All of the changes affect only the **Programmer's Guide**.

Here is a summary of the changes:

- A major new optimization called "cross-jumping" has been implemented.
- Register variables have been implemented, in a uniquely useful fashion.
- Any program using the 8087 or 80287 instructions is terminated at initialization time if the 8087 or 80287 is not present. Previously, this protection was available only for the coprocessor library routines. Furthermore, if an attempt is made to (erroneously) link code using 8087 or 80287 instructions with the emulator library, the linker produces a diagnostic that "_mw87_used" is undefined.
- Explicit instructions as to how to entirely remove the C I/O system are given.
- Toggle `Angle_include` allows one to instruct the compiler to treat `#include <...>` the same as `#include "..."`. This is useful when using standard header files that were not supplied with High C.
- Toggle `PCC_msgs` can turn off a subset of the compiler's warning messages. The conditions provoking the warnings are never detected by the standard UNIX PCC C compiler.
- Option `-ppo` and toggle `Print_ppo` allow one to capture preprocessor output in a separate file or in the listing file.
- Toggle `Print_protos` causes the compiler to automatically generate the new ANSI function prototype headers, to aid in upgrading programs to the emerging C standard.
- Compiler control `tmpi3` is no longer used.

Some of these items were documented in the "README" file in the 1.2a distribution but now have achieved full documentation status.

Now we detail how these new features affect the **Programmer's Guide**, and where possible give the exact text to appear in a future revised version of the **Guide**. For your convenience we have printed these addenda pages separately according to section so that they may easily be inserted into the 1.2 version of the Guide at the appropriate places.

Section 2

2.1 Invoking the Compiler

Binary dump utility. After the compiler has run, one can use the **bd** utility to dump the .OBJ OMF output file, printing out each OMF record and its contents. This utility is appropriate for finding out the size of various code and data segments emitted by the compiler. One needs to know about Intel OMF to understand the output in detail; order part number 121748-001, **8086 Relocatable Object Module Formats** from Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051.

The usage of **bd** is as follows, and the same instruction can be obtained by running **bd** with no parameters:

```
bd [-v] [-t TYPE] filespec ...
```

where there are one or more "filespecs" and each may have MS-DOS wildcards.

.OBJ files are dumped as OMF object modules.

.LIB files are dumped as Microsoft libraries.

.L86 files are dumped as Concurrent DOS 286 libraries.

All other files are dumped in simple hexadecimal. The following options apply:

-t TYPE = just dump records of type TYPE; e.g. "-t segdef".

-v = just dump invalid records (= -t ???) (for validating OMF files).

Section 3

3.2 Run-Time Libraries; Link Errors

If some of the program requires the use of the 8087 or 80287, in that it contains 8087 or 80287 native instructions, the program must be linked only with the coprocessor library. Attempting to link with the emulator library draws the linker error message "_mw87_used" undefined; this name is referenced in each object module making direct use of the 8087 or 80287. It is an error to link 8087/80287 in-line code with the emulator libraries. Such an error is automatically detected at link time by the missing name in the emulator libraries.

3.9 Minimizing Program Size

- Removing the C I/O subsystem

Even if none of C's I/O is used, some of the I/O system is still linked in. This portion does initialization and termination. It can be excluded from the link by providing the two externals _mwcfinit() and _mwcfterm(). Define these functions somewhere in the program to remove the library versions:

```
_mwcfinit() {}
_mwcfterm() {}
```

The savings is worth the effort.

Section 5

5.1 Command-Line Options

tmpi3. The **tmpi3** compiler control is no longer used. Although still accepted by the compiler, it has no effect.

ppo is a new command line option. If "**-ppo filename**" is given on the command line invoking the compiler, the preprocessor output is printed to "**filename**". If "**-ppo**" alone is given, the preprocessor output is printed to the standard output. In both cases the compiler terminates after preprocessing, i.e. compilation *per se* is not done. "**-ppo**" can be read "pre-process only" or "print preprocessor output". The preprocessor output is suitable for input to the compiler.

There is also a new toggle, Print_ppo, that causes preprocessed input to be printed (and sent to the listing file) when the toggle is On. With this toggle, it is possible to print what the compiler proper receives over a local area of source code. A use would be to turn the toggle On prior to a complex macro invocation and Off after it, to verify that the macro does what it should. The toggle is of course Off by default.

Section 7

7.2 System-Independent Toggles

Angle_include -- Default: On

This toggle means to process **#include <...>** in the standard fashion: look for the file in directories given in the <>-include Ipath, set up when configuring the compiler. Turning this toggle Off means to process **#include <...>** the same as **#include "..."**. The primary use of this is to avoid obtaining High C's standard include files when using those of another compiler; one can put the other compiler's include files on the standard High C Ipath instead of its <>-include ipath.

Optimize_xjmp -- Default: On

Enables the cross-jumping optimization. While an effective space-saving optimization that leaves execution time invariant, it slows the code generator a little and can produce code that is difficult to debug. See Appendix XJ for more information on the specifics of this optimization. Also see toggle Optimize_xjmp_space.

Optimize_xjmp_space -- Default: On

Enables cross-jumping optimization that saves space but always at the expense of time. This toggle takes effect only if Optimize_xjmp is also On. This optimization slows the code generator a little and can produce code that is difficult to debug. See Appendix XJ for more information on the specifics of this optimization. Also see toggle Optimize_xjmp.

PCC_msgs -- Default: Off

High C by default produces many warnings — code must be "squeaky clean" to get through the compiler without a warning. Some users have code that was designed with a PCC-style compiler (portable C on UNIX) that is not so demanding, and would prefer fewer prods from the compiler. Therefore, if toggle PCC_msgs is turned On, e.g. in the profile, the following warnings will *not* be emitted:

Function called but not defined.
Function return value never specified within function.
This "return" should return a value of type ttt
    since the enclosing function returns this type.
'=' used where '==' may have been intended.
Only fields of type "unsigned int" or "unsigned long int" are supported.
Bit length exceeds size of "unsigned int"; type changed to "unsigned long".
External function is never referenced.
Declared type is never referenced.

The next four messages are suppressed for global variables when PCC_msgs is On:

Variable is never used.
Variable referenced before set.
Variable is referenced but is never set.
Variable is set but is never referenced.

Print_ppo -- Default: Off

When On this toggle causes preprocessed input to be printed (and sent to the listing file). With this toggle, it is possible to print what the compiler proper receives over a local area of source code. A use would be to turn the toggle On prior to a complex macro invocation and Off after it, to verify that the macro does what it should.

Print_protos -- Default: Off

This toggle aids in the conversion of C programs to use the new ANSI prototype syntax derived from the C++ language. When this toggle is On, the compiler prints to the standard output a new, prototype-style function header for each function definition. For example, for the function definition

```
int f(x,y,z) int *x,z[]; double (*y)(); {...}
```

the compiler produces

```
int f(int *x, double (*y)(), int *z);
```

The old function header can then be replaced with the generated one.

There are some minor pitfalls in having the compiler automatically generate prototype headers. One is illustrated above: array parameters, according to the semantics of C, are converted to pointer parameters. Second, **enum** types are converted to their representation type (one of the **signed int** types). Finally, the compiler does not distinguish the type specifier **char** from the **signed-** or **unsigned-char** that **char** alone stands for. This means that for 8086 High C, both **char** and **unsigned char** are printed as **unsigned char**. On machines where the best default for **char** is **signed char**, both **char** and **signed char** are printed as **signed char**. The **enum** and **char** problems can be avoided by using **typedefs**, and using a **typedef** name for the parameter's type.

It may be desirable to use **-noobj** on the command line along with this toggle, to suppress compilation after the prototypes have been generated.

Print_reg_vars -- Default: Off

One can find out which variables were mapped to registers without looking at the code generated by the compiler by turning this toggle On.

7.3 System-Dependent Toggles

Auto_reg_alloc -- Default: Off

When On, the compiler automatically allocates variables to registers. The compiler weights variables used within loops more heavily than those not so used in making its decision which variables to allocate to registers; furthermore it will not allocate to registers variables that are used too infrequently. Automatic allocation allocates only variables of size 2 to registers; **chars**, for example, are not automatically allocated since it is not always best to place **chars** in a register. The reason for this is that preventing a register **char** from exceeding 127 (for signed **chars**) or 255 (for unsigned **chars**) is not inexpensive. **Auto_reg_alloc** has no effect unless toggle **Use_reg_vars** is On.

Return_32_in_BX_AX -- Default: Off

For Lattice compatibility, this toggle, when On, causes 32-bit quantities returned from functions to come back in AX:BX rather than DX:AX. This includes 32-bit pointers and long integers. In a future release this facility will probably be in the form of a calling convention rather than a toggle. (Another compiler option pertaining to Lattice compatibility is "Global_aliasing_convention" (Subsection 13.2). Since Lattice by default truncates externals to eight characters, use "**pragma** Global_aliasing_convention("%r:1:8");" in the profile to cause High C to do the same.)

Segmented_pointer_operations -- Default: On

(This is merely additional, clarifying documentation to what already exists in the present Guide.)

When the toggle is On, it is true that 32-bit pointer operations are generated by the compiler, including comparisons. But comparisons still *do not* normalize pointers before comparing. The programmer must normalize pointers himself.

Use_reg_vars -- Default: Off

When On, the compiler attempts to place variables of storage class **register** into one of the two machine registers SI or DI. Widespread use of register variables is not recommended for the 8086 family of architectures, but careful, selected use can be beneficial. Use_reg_vars is by default Off, since when initially porting a C program from a minicomputer environment where many register variable declarations are used, placing the variables in registers may actually have a detrimental effect on the 8086. For more information on register variables, see Section R (added between Sections 9 and 10).

Section 8

8.2 Floating-point Evaluation and Run-Time Libraries

(This supersedes some of the material in the penultimate paragraph of Subsection 8.2.)

The program may require the use of the 8087, either because some of its own modules contain 8087 instructions, or because routines from the coprocessor libraries have been linked in that in turn require the 8087. If so, the run-time start-up routine verifies that the 8087 exists and prints out an error message if it does not. Previously this detection only was given for any coprocessor library routines needing the 8087; now it also is given for user routines.

Under MS-DOS, one can find out which routines use the 8087 by searching for "87_used" in the .OBJ files. Use the **fgrep** utility program supplied in the MetaWare compiler distribution for MS-DOS: "**fgrep** -o 87_used *.obj" searches the objects for "87_used" and prints out the offset in the file containing "87_used".

If one tries to link any code requiring the 8087 with an emulator library, the linker will complain of the undefined name "_mw87_used". It is an error to link 8087 in-line code with the emulator library. Such an error is automatically detected at link time.

An entirely new Section R, logically fitting between Sections 9 and 10 of the present Guide, describes the implementation of register variables.

Section R
Register Variables

Both MetaWare's High C and Professional Pascal compilers support register variables. The implementation has a novel feature that separates it from all other 8086 compilers known to us, and that permits efficient use of register variables on an architecture that is hostile to them: the 8086 has few registers to dedicate to variables, and each register has a dedicated purpose.

Furthermore the compiler can automatically allocate variables to registers without requiring the programmer to supply the **register** storage class. In the sections that follow we present two different approaches to register variables. Then we discuss the porting of programs written for other machines relative to the two approaches. Next the automatic allocation of variables is described. Miscellaneous topics and a summary with tips for best usage of register variables conclude the presentation.

R.1 Register Variables — The Typical Approach

The typical implementation of register variables goes as follows. Registers SI and DI are earmarked as the two registers that can hold register variables — i.e. each function can have a maximum of two register variables. Each function that modifies SI and DI, whether it uses them for register variables or in any computation, saves SI and DI at procedure entry and restores them at procedure exit. Therefore each such function has a "**push si; push di**" at the beginning and "**pop di; pop si**" at the end.

The disadvantage of this strategy is that even if a program *never* uses register variables, all functions using SI and DI — for whatever purpose — must save and restore them. Thus an overhead is always incurred "just in case" somewhere, some function uses a register variable. The overhead is incurred for library routines as well as user program code.

The overhead is non-trivial, as a **push** and **pop** each take about 10 cycles on an 8088. In fact, a BYTE magazine analysis showed that programs using register variables can sometimes run slower. Although this is not always true, it has been confirmed in practice by some of MetaWare's customers using a compiler such as Microsoft C.

The strategy has its roots in compilers for mini- and mainframe computers, where register variables are the rule, and where compilers automatically allocate variables to registers, as MetaWare's own mini- and mainframe compilers do. In such an environment, the strategy is appropriate; the architecture often supports a save-multiple-registers instruction, e.g. IBM 370, or an automatic saving of registers on procedure call via a register mask, e.g. VAX 11/780.

On an 808x, which has neither of these two features, the heavy use of register variables is not appropriate.

R.2 Register Variables — The MetaWare Approach

MetaWare has solved the problem of this overhead with a feature unique to its 808x compilers. The programmer can choose whether or not functions that use SI and DI save and restore them as part of their prologue and epilogue. Let us call functions that do not save and restore SI and DI *non-preserving*, and functions that do *preserving*. In the standard approach, then, all functions are preserving. Now with non-preserving functions, how are registers then protected against damage via a call to a non-preserving function?

The answer is that when calling a non-preserving function, register variables are saved and restored by the caller. The example below illustrates the difference between the preserving and non-preserving approaches. Relevant assembly code is listed on lines starting with ";".

Preserving approach (the usual approach):

```
void f(int i) {
   ; push si          (Save.)
   ; push di            "
   ... some code that clobbers si and di ...
   ; pop  di          (Restore.)
   ; pop  si            "
   }

void main() {
   register int i,j;  /* i in si, j in di. */
   ...
   f(i);
   ; push si          (Pass i as parameter to f.)
   ; call f
   f(j);
   ; push di          (Pass j as parameter to f.)
   ; call f
   }
```

The non-preserving approach (featured in MetaWare compilers):

```
void f(int i) {
   ... some code that clobbers si and di ...
   }

void main() {
   register int i,j;  /* i in si, j in di. */
   ...
   f(i);
   ; push si          (Pass i as parameter to f.)
   ; mov  temp_si,si  (Compiler-coined temporary for si, alias i.)
   ; mov  temp_di,di  (Compiler-coined temporary for di, alias j.)
   ; call f
   f(j);
   ; push temp_di      (Pass j as parameter to f.)
   ; call f
   }
```

Now if "main" is the only function using register variables that calls "f", we have considerable savings in not unnecessarily saving and restoring SI and DI in "f" and all other functions in the program.

Therefore, the decision to use the preserving or non-preserving approach is made on the basis of frequency of register variables. If a program uses many register variables, the preserving approach is appropriate.

If a program uses register variables infrequently, the non-preserving approach is appropriate. A mixed strategy can also be used, since it is possible to specify on a function-by-function basis whether the function is preserving or not.

Functions can be declared as preserving by including _SAVE_REGS in their calling convention; by default, functions are non-preserving. For example:

```
pragma Calling_convention(_DEFAULT_CALLING_CONVENTION | _SAVE_REGS);
   void f(int i) {
      ; push si
      ; push di
      ... some code that clobbers si and di ...
      ; pop di
      ; pop si
      }                          /* Back to the default: */

pragma Calling_convention(_DEFAULT_CALLING_CONVENTION);
```

The default calling convention can be set to include _SAVE_REGS to get the preserving approach:

```
pragma Calling_convention(
            _DEFAULT_CALLING_CONVENTION | _SAVE_REGS, _DEFAULT);
            /* The default is now for all functions to save registers. */
```

For more information on the use of the calling convention pragma, see Section (13) *Externals*.

We at MetaWare believe that the non-preserving approach yields the best results on the 808x, when functions are chosen carefully in which to use register variables.

Consequently, the MetaWare run-time libraries do *not* save register variables. Thus functions such as "printf" should *not* be declared as preserving — e.g. do *not* declare them after the calling convention has been changed. Calling convention changes should be made *after* inclusion of library header files, not before.

One other thing to note is that the calling convention pragma applies only to functions that are defined. It does *not* apply to functions called but not defined. For example, if one writes "foo()" but never defines "foo", "foo" will be assumed to be non-preserving, no matter what has been done to the default calling convention. See Subsection 13.1 on undeclared functions.

A.3 Porting Programs Written for Other Machines

In light of this discussion, it is probably not desirable to take a program containing many register variables that was written for another computer and re-host it on the 808x along with all register variable declarations. The program may run slower with the register declarations than without. Therefore, the MetaWare 808x compilers *ignore* register declarations by default. To turn on the recognition of register declarations, use the toggle "Use_reg_vars". One can say "-on use_reg_vars" on the command line, or include an On and Pop pragma around the functions for which register variables are to take effect:


```
pragma On(Use_reg_vars);
   void f1() {
      register int i, j;    /* i, j will be allocated to registers. */
      ...
      }
pragma Pop(Use_reg_vars);              /* Restore to default (Off). */
   void f2() {
      register int i, j;    /* i, j will not be allocated to regs.  */
      ...
      }
```

R.4 Automatic Register Allocation

If register variables are really to be used everywhere, request that the compiler automatically allocate variables to registers. The compiler weights variables used within loops more heavily than those not so used in making its decision which variables to allocate to registers; furthermore it will not allocate to a register variables that are used too infrequently. Request automatic allocation by turning On the toggle "Auto_reg_alloc":

```
pragma On(Auto_reg_alloc);
   void f3() {
      int i, j; /* i and j get allocated to registers if justified. */
      ...
      }
```

Automatic allocation allocates only variables of size 2 to a register; **chars**, for example, are not automatically allocated since it is not always best to place **chars** in a register. The reason for this is that preventing a register **char** from exceeding 127 (for **signed chars**) or 255 (for **unsigned chars**) is not inexpensive. According to the formal definition of C being promulgated by the ANSI standard committee on C, the use of a **register** storage class must *not* change the semantics of a program. Here is a simple program that illustrates the difficulty:

```
main() {
   register unsigned char c; register int i;
   c = 250;
   for (i = 1; i <= 3; i++) {
      c += 10;
      printf("%d\n", c);
   } }
```

This program should print

```
250
4      (which is 260 mod 256)
14
```

but under a careless implementation of register variables, it might instead print

```
250
260
270
```

The code to add 10 to character c is not simply "**add si, 10**" (assuming c is allocated to **si**) but is instead "**mov ax, si; add al, 10; mov si, ax**". The "**add al, 10**" ensures that the addition does not overflow into the most significant byte

of c. Thus register chars are less efficient than might be expected, and so the compiler never automatically assigns chars to registers.

R.5 "Envelopes" for Frequently-Used Non-Preserving Functions

Suppose that in a particular region of a program heavy use is made of register variables and frequent calls are made to library functions (which, as we have said, are always non-preserving). To avoid the code space overhead of saving and restoring register variables across calls to library functions, consider hiding calls to frequently-called library functions in preserving functions dedicated to calling the relevant library functions.

For example, if malloc is called many times in the context of register variables, consider writing a preserving function "my_malloc" and replacing calls to malloc with calls to my_malloc:

```
void *my_malloc(unsigned amount) {return malloc(amount);}
```

Since my_malloc is a preserving function, SI and DI will be saved in its prologue and restored in its epilogue: a preserving function that calls a non-preserving one must save and restore both SI and DI in case the non-preserving function destroys them.

R.6 Miscellaneous

Variables that were mapped to registers can be found without looking at the code generated by the compiler by turning on the toggle Print_reg_vars: either use "-on Print_reg_vars" on the command line, or include "pragma On(Print_reg_vars);" in the program.

R.7 Effect of Dedicated Instructions on the Use of SI and DI

Certain 808x instructions require the use of SI and DI, such as byte-moving or -scanning operations. The former arise in structure assignments; the latter when using MetaWare High C's built-in byte scanning operations. The required use of SI and DI cause a conflict when they are allocated to variables. In these cases the compiler stores the register variables into temporaries, reloading them back into registers upon the next usage.

R.8 Summary; Pointers and Tips

Here are the salient facts that should be considered when using register variables:

1. There are two approaches to using register variables: the standard "preserving" approach, or the MetaWare "non-preserving" approach. MetaWare compilers support both approaches, but by default functions are non-preserving.

2. Use the preserving approach only if the intent is to use register variables frequently. Use the non-preserving approach otherwise.

3. A preserving function that modifies SI and DI saves and restores them in its prologue and epilogue, respectively. Any call to a non-preserving function is

considered to be a modification of SI and DI, so that preserving functions calling non-preserving functions always save and restore SI and DI.

4. When a call is made to a non-preserving function from a function having register variables, the variables are saved before the call and restored the next time that they are needed. Thus, successive calls to non-preserving functions may require saving and restoring only once.

5. Use the calling convention pragma with _SAVE_REGS to specify preserving functions. If all functions are to be preserving, set the default calling convention to contain _SAVE_REGS.

6. Calling convention pragmas apply only to defined functions. Functions called but not defined are always non-preserving — the default. Thus, when using the calling convention to establishing preserving functions, always declare all functions before using them.

7. Library functions are non-preserving. This includes "hidden" library functions such as those implementing 32-bit arithmetic and floating-point emulation. It is strongly advised that register variables not be used in the presence of much long or emulated-floating-point arithmetic.

8. *DO NOT DECLARE* library functions after changing the calling convention to include _SAVE_REGS. Doing so erroneously claims that library functions are preserving. Likewise, *do not* include library header files after changing the calling convention. There may be declarations of library functions embedded in functions that will have to be relocated if the calling convention is changed.

9. Variables that are not two bytes in size may not pay off when placed in a register. Use registers for such variables only when the number of references to them considerably exceeds the number of assignments to them. The compiler never automatically allocates such variables to registers.

10. To make register variable declarations take effect, it is necessary to turn On the toggle Use_reg_vars.

11. The compiler automatically allocates variables to registers when the toggle Auto_reg_alloc is turned On.

12. It is possible to find out which variables were allocated to registers, whether automatically or not, by turning On the toggle Print_reg_vars.

13. Both local variables and function parameters can be register variables.

14. We would appreciate feedback on this unique approach to register variables. Especially consider whether it is preferable to have the library save and restore register variables, even though a program might contain no use of register variables anywhere, and even though saving and restoring increases overhead in time and space during the execution of library functions.

Section 10

10.3 The Stack Frame

A local variable is addressed by some negative displacement off of the BP register, except when that local variable has been allocated to a register — either SI or DI. In the latter case the variable occupies no space on the stack. A local variable can be allocated to a register by using the **register** storage class in conjunction with the Use_reg_vars toggle.

Parameters are addressed with a positive displacement off of the BP. When a parameter is placed in a register, it is addressed in this way only once, at procedure prologue, in loading the parameter into a register. Thereafter the parameter is referenced in the register. A parameter can be allocated to a register by using the **register** storage class in conjunction with the Use_reg_vars toggle.

Be warned that use of library functions setjmp and longjmp can produce unpredictable results in the context of register variables. longjmp can cause an arbitrary number of function returns, none of which restores SI or DI. See the Library Reference Manual for more on setjmp and longjmp.

Section 11

11.2 Prologues and Epilogues

If a function is declared with _SAVE_REGS in its calling convention (see Section 13.1; by default a function does *not* have _SAVE_REGS in its calling convention), the function will save registers SI and DI at entry and restore them before returning if the function in any way modifies those two registers.

A function is considered to modify SI or DI if either SI or DI is used in any code generated for the function, or the function calls another function that does not have _SAVE_REGS in its calling convention — i.e. the other function may in turn destroy SI and DI.

The function saves SI and/or DI immediately after allocating its own local stack frame (the `sub sp, framesize` instruction).

11.5 Saving Registers on a Call

Consider a function F', using variables in registers SI and DI, calling a function F that does not save and restore SI and DI; i.e. F does not have _SAVE_REGS in its calling convention. F' is forced to save its own register variables and restore them after the call to F.

Heavy use of register variables in the presence of calls to functions not having _SAVE_REGS in the calling convention is not recommended, since the save/restore overhead may well defeat the intended gain in efficiency with the use of register variables. See the new Section (A) *Register Variables.*

The restore is not always done immediately after the return from the call; it may be delayed, especially if several calls appear sequentially.

Section 12

12.5 The Effect of Optimizations on Debugging

Most compiler optimizations do not severely affect the order in which code is generated with respect to the original order of statements in the program. Therefore, it is generally possible to use a symbolic, line-oriented debugger, and it is possible to keep track of code generated when debugging in assembly language.

There is one class of optimizations that can severely reorder code, however, and thoroughly confuse debuggers and humans reading assembly-level debugger output. This class is discussed in Appendix (XJ) *Cross-Jumping*. The optimization is turned On by default, and we recommend that it be turned Off, if debugging becomes a problem. See Appendix *Cross-Jumping* for more details.

Section 13

13.1 Interfacing to Other Languages

The following calling-convention pre-defined constant is added to the existing list:

Name	*Semantics*
_SAVE_REGS	The function saves registers SI and DI if it modifies either of them, either by direct use of SI or DI in an instruction, or by calling a function that does not have _SAVE_REGS in its calling convention and that therefore could destroy SI or DI. _SAVE_REGS is *not* included in the _DEFAULT_CALLING_CONVENTION. Nor do any of the High C library functions include _SAVE_REGS in their calling convention.

Appendix XJ
Cross-Jumping Optimizations

MetaWare compilers now support a major new optimization that can usually obtain a 2—5% reduction in code size and is often accompanied by a decrease in execution time. The optimization is known as "cross-jumping". It, along with the two toggles that control it, is explained here.

Consider the following source code:

```
                if (!eof) read(&buf,&cnt,512);              /* Code C.  */
/* L: */        while (cnt > 0) {
                   write(&buf,cnt);
                   if (!eof) read(&buf,&cnt,512);           /* Code C'. */
                   }     /* Implicit jump back to the implicit label L. */
```

The compiler can improve the code size of this program without any loss in execution speed by effectively re-writing the code as:

```
   Top:         if (!eof) read(&buf,&cnt,512);           /* Code C = C'. */
/* L: */        if (cnt > 0) {
                   write(&buf,cnt);
                   goto Top;
                   }
```

The optimization involves the recognition of some code C immediately preceding a jump j to some label L, where some code C' identical to C immediately precedes L. The transformation consists in deleting C and replacing j with a jump to C' instead:

```
                some code C     .               jmp L'
                jmp L           =>
                ...                             ...
                some code C'        L':         some code C = C'
   L:           ...                 L:          ...
```

This optimization is called "cross jumping" or "tail merging" in the compiler literature, since it was first invented to handle common code at the ends of the arms of conditional statements, and was effected by jumping across from one arm to the other, i.e. by merging the tails of the two arms. It is surprisingly effective and always saves code space while never giving up execution speed.

Here we include another optimization under that name as well. The second optimization is even more effective but gains (sometimes considerable) code space in trade for a small loss of speed. Consider the program fragment

```
if (buf[cnt]==0) g(&buf);
   else if (buf[cnt]=='\n') {buf[cnt] = 0; g(&buf);}
   else ...
```

The compiler effectively transforms this into

```
if (buf[cnt]==0) goto L';
   else if (buf[cnt]=='\n') {buf[cnt] = 0; L': g(&buf);}
   else ...
```

Here, both occurrences of "g(&buf);" precede a jump to the statement following the entire conditional. One of the instances of "g(&buf);" is replaced

with a jump to the other, saving the code space for the call to g at the expense of inserting an additional jump. Opportunities for this kind of optimization are even more frequent than the standard cross-jumping optimization. In general the optimization can be depicted as follows:

```
        some code C                    jmp L'
        jmp L
        ...                            ...
        some code C'    =>  L':        some code C + C'
        jmp L                          jmp L
        ...                            ...
L:      ...                     L:     ...
```

Both optimizations are turned On by default. Both may be disabled by turning Off the toggle Optimize_xjmp, with either "-off Optimize_xjmp" on the compiler execution line, or including "pragma Off(Optimize_xjmp);" in the program. The second of the two optimizations can be disabled by turning Off the toggle Optimize_xjmp_space, so named because the second optimization saves space but always increases execution time.

During the development phase of a project it may be desirable to turn Optimize_xjmp Off. The reason is that the optimization can cause such a contortion of code that using debuggers, whether assembly-language level or line-oriented symbolic, is difficult. As a case in point consider the following program, which compares the fields of two different structures to see if they are the same:

```
union {
   struct {int x, y;}            f1;
   struct {int a, b, c;}         f2;
   struct {int e, f;}            f3;
   struct {int g, h; int i[10];} f4;
   } u1, u2;

int f(i) int i; {
   switch(i) {  /* What kind of structure to compare? */
      case 1: return u1.f1.x == u2.f1.x && u1.f1.y == u2.f1.y;
      case 2: return u1.f2.c == u2.f2.c &&
                     u1.f2.a == u2.f2.a && u1.f2.b == u2.f2.b;
      case 3: return u1.f3.e == u2.f3.e && u1.f3.f == u2.f3.f;
      case 4: return u1.f4.g == u2.f4.g &&
                     memcmp(u1.f4.i, u2.f4.i, sizeof(u1.f4.i)) != 0;
      case 5: return u1.f4.h == u2.f4.h &&
                     memcmp(u1.f4.i, u2.f4.i, sizeof(u1.f4.i)) != 0;
   } }
```

Here cases 1 and 3 are recognized as being identical, and matching the tail end of case 2. Furthermore cases 4 and 5 share a common tail. Compiling the code produces the following tightly-coded result that surpasses the usual patience of even a skilled assembly-language programmer in optimizing:


```
;           switch(i) { /* What kind of structure to compare? */
        mov   bx,4[BP]
        dec   bx
        cmp   bx,4
        jnbe  006a
        shl   bx,1
        jmp   word ptr cs:.L0013[bx]
.L0013:
        dw    .L0026
        dw    .L001d
        dw    .L0026
        dw    .L003a
        dw    .L0043
;           case 1: return u1.f1.x == u2.f1.x && u1.f1.y == u2.f1.y;
;           case 2: return u1.f2.c == u2.f2.c &&
.L001d:
        mov   ax,@test+4
        cmp   ax,@test+28
        jne   0066
;                               u1.f2.a == u2.f2.a && u1.f2.b == u2.f2.b;
;           case 3: return u1.f3.e == u2.f3.e && u1.f3.f == u2.f3.f;
.L0026:
        mov   ax,@test
        cmp   ax,@test+24
        jne   0066
        mov   ax,@test+2
        cmp   ax,@test+26
        jne   0066
        jmp   0062
;           case 4: return u1.f4.g == u2.f4.g &&
.L003a:
        mov   ax,@test
        cmp   ax,@test+24
        jmp   004a
;                               memcmp(u1.f4.i,u2.f4.i,sizeof(u1.f4.i)) != 0;
;           case 5: return u1.f4.h == u2.f4.h &&
.L0043:
        mov   ax,@test+2
        cmp   ax,@test+26
.L004a:
        jne   0066
        mov   ax,20
        push  ax
        mov   ax,offset @test+28
        push  ax
        mov   ax,offset @test+4
        push  ax
        call  memcmp
        add   sp,6
        and   ax,ax
        je 0066
.L0062:
        mov   al,1
        jmp   0068
.L0066:
        sub   ax,ax
```



```
.L0068:
        sub     ah, ah
.L006a:
        pop     bp
        ret
;                                   memcmp(u1.f4.i, u2.f4.i, sizeof(u1.f4.i)) != 0;
;       } }
```

Compare this code with that generated by other compilers and generally it will be found to be much smaller.

In summary:

1. Cross-jumping is an amazingly effective optimization.
2. Toggle "Optimize_xjmp" is set On by default and turning it Off disables all cross-jumping.
3. Toggle "Optimize_xjmp_space" is On by default and turning it Off disables cross-jumping optimization that decreases space at the expense of time.

The cross-jumping optimization adds perhaps 20% to the execution time of the code generator phase of the compiler, thus perhaps 3% overall.

