SECTION 19

Compiler Notes

This section supplements Section 21 of the Explorer Lisp Reference manual with some additional information that might be of interest to system programmers.

19.1 COMPILE-TIME PROPERTIES OF SYMBOLS

When symbol properties are referred to during macro expansion, properties defined in a file should be in effect for the compilation of the rest of the file. This does not happen if GET and DEFPROP are used, because the DEFPROP is not executed until the file is loaded. Instead, you can use GETDECL and DEFDECL. These are normally the same as GET and DEFPROP, but during file-to-file compilation, they also refer to and create declarations.

SYS:FILE-LOCAL-DECLARATIONS

During file-to-file compilation, the value of this variable is a list of all declarations that are in effect for the rest of the file. Macro definitions, DEFDECL's, PROCLAIM's and special declarations that come from DEFVAR's are all recorded on this list.

GETDECL function-spec property

This function is a version of GET that allows the properties of a function to be overridden by local declarations.

If LOCAL-DECLARATIONS or SYS:FILE-LOCAL-DECLARATIONS contain a declaration of the following form:

```
(property function-spec value)
```

GETDECL returns value. Otherwise, GETDECL returns the result of the following form:

```
(function-spec-get function-spec property)
```

The GETDECL function is typically used in macro definitions. For example, the SETF macro uses GETDECL to obtain the SETF property of the function in the expression for the field to be set.

PUTDECL function-spec property value

The PUTDECL function causes (GETDECL _function-spec_ _property_) to return _value_.

The PUTDECL function makes an entry on SYS:FILE-LOCAL-DECLARATIONS of the following form:

```
(property function-spec value)
```

This form stores _value_ where GETDECL can find it; but if PUTDECL is called during compilation, it affects only the rest of that compilation.

DEFDECL symbol property value

When executed, DEFDECL resembles PUTDECL except that the arguments are not evaluated. This special form is usually the same as DEFPROP except for the order of the arguments.

Unlike DEFPROP, when DEFDECL is encountered during file-to-file compilation, it is executed, creating a declaration that remains in effect for the rest of the compilation. (The DEFDECL form also goes into the xld file to be executed when the file is loaded). The DEFPROP special form would have no effect whatever at compile time.

The DEFDECL special form is often useful as a part of the expansion of a macro. It is also useful as a top-level expression in a source file.

Consider the following form:

```
(DEFDECL FOO SETF ((FOO X) . (SET-FOO X SI:VALUE)))
```

The preceding form in a source file allows the following form to be used in functions in that source file; and by anyone, once the file is loaded:

```
(SETF (FOO ARG) VALUE)
```

COMPILER:*LOCAL-DECLARATIONS-SPECIFIERS*

This variable is a list of declaration specifier symbols for which the compiler will push the declaration onto the LOCAL-DECLARATIONS list for access by GETDECL. Users can define new declarations by pushing the name onto this list. Compatibility note: in Explorer Releases 1 and 2, all declarations were pushed on LOCAL-DECLARATIONS, but Release 3 is more selective for the sake of efficiency.

19.2 DECLARATIONS

The following declaration specifiers are used internally in addition to those documented in section 13 of the Explorer Lisp Reference manual.

:EXPR-SXHASH number

At the beginning of the body of a DEFMACRO, DEFSUBST, or inline function, a (DECLARE (:EXPR-SXHASH <number>) may be used to specify the hash code that will be recorded for the macro. This hash code is used for giving warnings when loading a file that uses macros whose definitions have changed; it is normally computed by hashing the definition. The :EXPR-SXHASH declaration can be used to cause a slightly modified version of a macro to have the same hash code as the previous version (obtained from the debug-info) in order to suppress these warnings.

COMPILER:TRY-INLINE function-name ...

This is similar to an INLINE declaration except that the inline expansion will be used only if, after optimization, it is not significantly larger than the original function call. This is useful for functions that can be optimized down to something trivial when some of the arguments are constants or known to be of a particular type, but are too big to be included inline in the general case.

SYS:DOWNWARD-FUNCTION

The declaration (DECLARE SYS:DOWNWARD-FUNCTION) can be used at the beginning of the body of a LAMBDA expression or local function to inform the compiler that the lexical closure which is being created is only being passed downward and there will be no references to it after execution leaves the context in which it was created. This enables the compiler to use the instruction MAKE-EPHEMERAL-LEXICAL-CLOSURE instead of MAKE-LEXICAL-CLOSURE, thereby saving some run-time overhead. Note that in many cases, the compiler is able to figure this out for itself without needing the declaration, and that incorrect use of the declaration could cause severe problems.

19.3 XLD FILES

All xld files are composed of 16-bit bytes. The first two bytes in the file contain fixed values, which are present so that the system can tell a proper xld file. The next byte is the beginning of the first <u>group</u>. A group starts with a byte that specifies an operation. It can be followed by other bytes that are arguments.

Most of the groups in an xld file are present to construct objects when the file is loaded. These objects are recorded in the <u>fasl-table</u>. Each time an object is constructed, it is assigned the next sequential index in the fasl-table. The indices are used by other groups later in the file to refer back to objects already constructed.

To prevent the fasl-table from becoming too large, the xld file can be divided into <u>whacks</u>. The fasl-table is cleared out at the beginning of each whack.

The other groups in the xld file perform operations such as evaluating a list previously constructed or storing an object into a symbol's function cell or value cell.

19.4 OPTIMIZATION

Besides the functions COMPILER:ADD-OPTIMIZER and COMPILER:OPTIMIZE-PATTERN which are described in section 21.8 of the <u>Explorer Lisp Reference</u> manual, the following functions may be used to specify optimizations:

COMPILER:FOLD-CONSTANT-ARGUMENTS function-name

Tells the compiler that if it sees a call to the designated function in which all of the arguments are constants, then it can call the function at compile-time and replace the function call with a QUOTE form containing the resulting value. This also implies that the function has no side-effects, so calls can be deleted if their value is not used.

COMPILER:DEFCOMPILER-SYNONYM &quote function synonym-function

Both arguments should be symbols, and are not evaluated. When the compiler sees the first argument used as the name of a function to be called, it will compile code to call the function named by the second argument instead.

COMPILER:OPTIMIZE-STATUS

This function can be called to find out the current values of the OPTIMIZE switches. The value returned is in the form of a declaration specifier so that one can do:

```
(SETQ SAVE-OPT (COMPILER:OPTIMIZE-STATUS))
...
(PROCLAIM SAVE-OPT)
```

to save and restore the switches.

19.5 COLD-LOAD ATTRIBUTE

Putting the attribute COLD-LOAD:T in the mode line of a file lets the compiler know that the file is part of the minimal kernel, or in other words, it is loaded by Genasys into the cold band. This knowledge is used in the following ways:

1. The compiler can warn you about using features that Genasys does not support. For example:
   a. Genasys can't handle load-time evaluation of constants [reader macro #,].
   b. (PROCLAIM '(TYPE ...)) doesn't work until after the compiler is loaded.
2. The compiler can give special handling where needed. For example, package commands such as EXPORT that don't supply an explicit package argument are automatically given one by the compiler because defaulting to *PACKAGE* is not appropriate during "crash-list" evaluation.
3. The compiler can warn you about using functions that are not in the cold load. Anyone who has ever had to debug a band that won't boot because it is trying to call an undefined function should appreciate the value of this. More about this below.
4. Since the COLD-LOAD attribute identifies the file as part of the kernel, style warnings for use of internal functions are suppressed. For example, you can use microcoded functions such as MEMQ and FIND-POSITION-IN-LIST without getting a complaint that they are obsolete.
5. Top-level forms which are going to be evaluated at load time will be fully macro-expanded at compile time in order to minimize the amount of work done in evaluating the "crash-list" when the band is booted and to avoid potential problems with macros that are undefined or that use things that aren't initialized yet.

Warnings about use of functions that are not in the cold load are implemented by checking the source file pathname of each function that is called to see if it has the COLD-LOAD attribute also. Thus, for these warnings to be meaningful, it is essential that all the files that are part of the cold load have the attribute. To avoid getting irrelevant warnings, functions that are not really needed in the cold load should be in separate files from those that are. Currently, these warnings are suppressed in COMPILE-FILE (they appear only when compiling in an editor buffer or with SAFETY>1) to avoid getting a great many warnings during system builds. For cases where you really do need to reference a function that won't really be called in the cold band, you can suppress the warning by binding the INHIBIT-STYLE-WARNINGS-SWITCH flag. For example:

```
(IF (FBOUNDP 'FOO)
    (COMPILER-LET ((INHIBIT-STYLE-WARNINGS-SWITCH T))
      (FOO X)))
```