MACRO-32 Type Checker

Paul Houlihan, OpenVMS Engineering

This report describes DCL command files that support the checking of the embedded data type in MACRO-32 field names against an opcode's expectation for each argument. The benefit of this data type checking is increased productivity and quality when programming in MACRO-32.

March 30, 1991

Contents

1 Introduction

Benefits/Features

The MACRO-32 Type Checker is a DCL command file that checks the data types embedded in a MACRO-32 argument field name against the opcode's expectation for each argument. The following benefits/features are provided by the MACRO-32 Type Checker:

- IMPROVED PRODUCTIVITY - A programmer can immediately access data type conflicts on a newly written or modified MACRO-32 module instead of painstakingly extracting each bug through manual debugging. A potential problem with type checking is that too many warnings can be produced to process in a reasonable amount of time. The Type Checker addresses this problem by providing options to control the different types of checking performed and by giving complete control over the type checking rules. Another unmanageable problem is sifting out new warnings that result from minor changes to a module from previously existing and verified warnings. A command file exists that makes identifying the different warnings in two listing files a manageable task.
- IMPROVED QUALITY - MACRO-32 code on infrequently executed code paths can be checked for common failures instead of waiting until a field test site or customer discovers the bug. It should be possible with this tool for an engineer to never ship or even checkin another bug resulting from the confusion over a VIELD ($V_) or MASK ($M_) field! The tool can be used by developers to verify changes or after source checkin by anyone interested in monitoring one dimension of quality in MACRO-32 code. The Type Checker also encourages the clean up of field naming and usage in order to reduce the number of warnings generated and to improve the type checking which results in more maintainable code.
- EASY TO USE - No complicated learning curve is involved. The Type Checker requires no tailoring, just provide a VAX MACRO or MACRO Compiler listing file and the Type checker returns a copy of the listing file with simple descriptive warnings embedded.
- FLEXIBILITY - A powerful optional aspect of the Type Checker is the ability to tailor the checking rules to the specific conventions of a facility to support more rigorous checking or reduce the noise of false warnings. This flexibility helps the engineer customize the type checking to their code, focusing in quickly on problems without wasting time examining excessive false warnings.

Overview

The MACRO-32 field naming conventions in the *VMS Internals and Data Structures (IDSM)* recommends a format of:

<data_structure_name>$<data_type>_<field_name>

The data_type field can be used to check against the opcode's argument expectations. For example the opcode MOVZBL is expecting a byte field ($B_) for the first argument and a longword field ($L_) as the second argument. The MACRO-32 Type Checker is a simple text processor that is able to parse the opcode and the text of any arguments. You define the strings that identify a data type of a field, for example, "$B_" identifies a byte data type field. You also define for each opcode the data type(s) that are expected for each argument. The Type Checker then issues warnings whenever an argument of the wrong data type is detected.

In addition the Type Checker checks for other common failures, such as missing #-signs or indexing errors. The problems being looked for by the Type Checker were suggested by OpenVMS engineers and by a perusal MACRO-32 bugs in OpenVMS VAX V5.0, V5.2 and V5.4.

The remaining sections of this Type Checker documentation cover current checking capabilities, the files that comprize this tool and the warnings and errors that can be generated. The files are:

- TYPCHK.DEFINITIONS - contains the editing rules and options.
- TYPCHK.COM - is the DCL command file that performs the type checking. It takes as input a VAX MACRO Listing file or a MACRO Compiler Listing file along with TYPCHK.DEFINITIONS file and outputs a copy of the listing file with warnings embedded.
- TYPCHK_DIFF.COM - is the DCL command file that can be used to DIFF the Type Checker warnings of two files.
- TYPCHK.PS - this documentation

Thanks to Linda Duffell for encouraging me in doing this project and to Dick Stockdale for field testing and making many suggestions on the Type Checker.

2 TYPCHK.DEFINITIONS

A powerful feature of the Type Checker is that all checking rules and selected options are contained in a simple text file called TYPCHK.DEFINITIONS. The checking rules describe the data type expectations for all opcode arguments and the options select among checking rules that are hardwired into the TYPCHK.COM. The definition file is input along with a listing file on each execution of TYPCHK.COM. The checking rules and options are specified by a few simple statements. The syntax of these statements is embedded in the comments of the TYPCHK.DEFINITIONS file and is not repeated here.

To facilitate facility-specific versions of TYPCHK.DEFINITIONS, TYPCHK.COM performs three steps in trying to locate the TYPCHK.DEFINITIONS file:

1. The current default directory is searched and the file TYPCHK.DEFINITIONS is opened if found.
2. If the logical name TYPCHK$DEFINITIONS exists then the file pointed at by this logical is read.
3. The default master copy of the definition file, not tailored to any specific facility, is used which is simply hardcoded to be in the following directory: DOCD$:[TOOLS]. To change the default location of the master copy on your system merely edit the setting of the DCL symbol MASTER_LOC in TYPCHK.COM.

Once the definition file is located, the rules and options are read and the type checking is performed.

The remainder of this section gives a short description of each TYPCHK.DEFINITIONS statement:

- DEFINE_TYPE - This statement defines the strings that identify the data type of an argument. An example of this statement is:

```
DEFINE_TYPE             BYTE        $B_,$GB_,$AB_,_B_,_GB_,_AB_
```

  — In this DEFINE_TYPE statement example, the BYTE data type is defined as an argument containing the strings $B_, $GB_, etc. These strings are referred to as Data Type ID Strings. I have used the data naming conventions suggested by the IDSM in the master copy of TYPCHK.DEFINITIONS. The usage of most data types was obvious but some types like TEXT or ADDR (the address data type) were difficult to predict. For example, ADDR generally has been assigned wherever LONG is used as a 32 bit entity (ie. in CMPL or CLRL). But it would be rare to be using an address field in an arithmetic operation or with a bit manipulation instruction, so ADDR is not used for such types of instructions. The data types were defined to minimize pointless noise given OpenVMS conventions. Other suggested IDSM data types like RECORD, NUM_STRING and VARIABLE were unpredictable and were left out.

  — If your facility has naming conventions that vary from the IDSM suggested conventions then the conventions can be setup with the DEFINE_TYPE statement. This also provides an opportunity to make consistent the naming conventions within your facility to reduce the false warnings and improve error detection.

- Note that the following special data types ID strings exist:
  - x *NO_CHECK* - is a special Data Type ID String that specifies that no checking is to be done on this data type.
  - x *LABEL* - is a special Data Type ID String that is defined to be the format of a valid MACRO-32 numeric label.

- DEFINE_OPCODE - This statement defines the data type of each argument expected for each VAX opcode and EVAX builtin. More than one data type for a given argument can be specified by separating them with commas and enclosing them in parens. The object of defining multiple data types for an argument is not to suppress all warnings. Your goal should be to have enough data types to cover common safe usage and avoid excessive warnings but still generate enough warnings to catch errors. Thus getting some warnings on a module is OK and the TYPCHK_DIFF.COM command file exists to help you detect changes in warnings between different versions of a module.

  Literals can appear frequently in place of expected data types. For example, the MOVL opcode expects the first argument to be a longword but a common occurrence is "MOVL   #BDB$C_VALID,R0". Rather than generated excessive warnings on use of such literals, if the expected data types from the DEFINE_OPCODE statement are not literal data types then the Type Checker substitutes the following data types: CONSTANT, SIZE, MASK. Thus for literal arguments not expected to be literal, these three data types are the expected data types output on any data type mismatch warning.

  Macros can be type checked as long as the arguments are standard MACRO-32 arguments using the DEFINE_OPCODE Statement. The macro name is used in place of the opcode name. See the example for $REMQHI in the master copy of the definition file.

  Note that an argument may have an optional 2 character suffix that describes the access type and data size of the argument. The first character is the access type which is used by the FLAG_LITERAL_WRITE_ADDRS option (see the DEFINE_OPTION description below) to determine arguments that are used as the address of a write. The second character is the data size which is used by the Indexing Flags to determine the size of indexing used on this argument. In the following example, the first argument is Read-only and is a data size of Byte, the second argument is Modify (both read and written) and is also a data size of Byte. Valid suffix characters are defined in the definition file comments for the DEFINE_OPCODE statement.

  ```
  DEFINE_OPCODE   BICB2   (BYTE,MASK,CONSTANT)-RB          BYTE-MB
  ```

- DEFINE_EXCEPTION - You can specify an exception for fields that disobey conventions but have been determined to be safe, in order to eliminate warnings on these fields. Any argument containing one of these exception strings will never be the target of any data type mismatch warning message that triggers as a result of violation of the checking rules established by the DEFINE_OPCODE statement. An example of this statement is:

  ```
  DEFINE_EXCEPTION          CTL$GQ_RMS_RUREC_LKSB
  ```

- DEFINE_TYPE_MODE - This statement allows you to specify a field data types that must or must not be used with certain addressing modes. Addressing modes are hardwired into TYPCHK.COM and currently only LITERAL addressing mode is defined. Other addressing modes can be added if there are good uses for them. In the master copy of the definition file, the CONSTANT, SIZE, MASK and VIELD data types must be used in a literal addressing mode or the argument will be flagged. Also ANY_TYPE data type must not be a literal. An example of this statement is:

```
DEFINE_TYPE_MODE          CONSTANT    LITERAL
```

- DEFINE_OPTION - Checking rules that have been hardwired into TYPCHK.COM are selectable as options on the DEFINE_OPTION statement. An example of this statement is:

  ```
  DEFINE_OPTION             FLAG_INTEGERS,FLAG_LOCAL,FLAG_INDEXING_MISMATCHES
  ```

  The following options are currently defined:

  — FLAG_INTEGERS - Flag all integer-only arguments which are integer values not preceded by a #-sign or associated with a register. This is to catch the missing #-signs on arguments. The flag is selected by default in the master copy of the definition file.

  — FLAG_LOCAL - Flag local module symbols used in arguments. Local symbols often do not follow the naming convention of embedding the data type overtly in the symbol name. By default in the master copy of the definition file, this option is selected as it is useful to check field size assumption on these local symbols. However there may be facilities where this option results in an excessive number of warnings. A local symbol is a field name that contains alpha-numeric or underscore characters but no "$". Before checking in a module, at least one pass should be made with this option selected to verify the usage of these local symbols.

  — FLAG_INDEXING_MISMATCHES - Flag all arguments that use indexing where the data size used in indexing does not match the data type of the field. The data size used in indexing is 1-Byte, 2-Word, 4-Longword or Float, 8-Quadword or D-Float, G-Float and 16-Octaword or H-Float. This indexing data size is specific to each opcode argument. For simple opcodes like MOVL, both arguments use longword indexing. For more complicated opcodes like EXTV, the first argument does longword indexing, the second and third arguments use byte indexing and the final argument uses longword indexing.

    To catch unintended indexing, the data size for indexing is specified in a suffix for each argument on the DEFINE_OPCODE statement. The Type Checker data types that are valid for a given data size are:

Data Size	Expected TYPCHK Data Types
BYTE	BYTE, TEXT
WORD	WORD
LONG	LONG, ADDR, PTR_L
QUAD	QUAD, PTR_Q
OCTA	OCTA

    If the data type embedded in a field used in indexing is not an expected data type then a warning is issued. This option is selected by default in the master copy of the definition file. Note arguments with an access type of "B"ranch are not flagged by this option. This was done to reduce noise as indexing on such arguments always use byte indexing and aren't usually prone to errors.

  — FLAG_NODT_INDEXING - Flag all arguments that are using indexing but have no embedded data type as there is no field name (ie. (R1)[R2]). This option flags for manual checking, instances of indexing that can't be automatically type-checked. By default, this option is not set in the master copy of the definition file but one pass should be made with this option selected to verify all indexing. Note arguments with an access type of "B"ranch are not flagged by this option.

  — FLAG_UNSIGNED_BRANCHES - Flag all unsigned branches (BGTR, BGEQ, BLSS, BLEQ) that follow a byte or word opcode. This option attempts to locate branch opcodes that should be unsigned as they are following an operation on an unsigned field. A typical example is a size field that is meant to be a full byte or word or longword. The use of one of the signed branches listed above may yield unintended results when the size field is large enough to set the highest order bit, the sign bit! Typically overflow of an unsigned field occurs on a byte or word field. Unsigned longword or quadword fields could also have this problem but this is more rare and so these fields were not included to cut down on noise. Note that BEQL and BNEQ are not involved in the option as the signed and unsigned forms of these opcodes are identical. This option is selected by default in the master definitions.

— FLAG_LITERAL_WRITE_ADDRS - Flag all instructions that write values to a literal address which is normally unintended. An argument that is used for writing is assigned a "W"rite or "M"odify as the access type in an argument suffix on a DEFINE_OPCODE statement. This option is selected by default in the master definitions.

— FLAG_IMPLICIT_REGISTER_USE - Flag all opcodes that implicitly destroy (or return information in) registers other than AP, FP, SP or PC. Verifying that each destroyed register is properly saved and restored can often prevent a common bug. The message indicates which registers are destroyed by the opcode in question. This option is selected by default in the master definitions.

— ASSUME_SUPPRESS- ASSUME statements can suppress data type mismatch warnings that result from a violation of the checking rules described in the DEFINE_OPCODE statements. If this option is selected then any field which is included in a recent ASSUME statement prior to a data type mismatch will have the warning suppressed. Note no other options or warning messages are affected by this option. If the option is not selected then ASSUME statements are ignored. Note only the last five ASSUME statement prior to the mismatch are checked. The option is not selected by default in the master definitions since ASSUME statements can contain outdated assumptions. It is a good idea to make at least one pass with this option not selected before checkin of a new or significantly changed module to verify all data type mismatches are correctly protected by ASSUME statements.

— FULL_TYPCHK_FILE - If this option is selected then the file produced by TYPCHK.COM with the embedded warnings will be a complete copy of the original MACRO listing file. Without this option, the created TYPCHK file terminates after the last instruction omitting the symbol table or Alpha machine code listing. By default this option is not selected in the master definitions to save disk space.

— FLAG_REGISTER_ONLY_ARGS - Flags all arguments that are merely registers without a field name to permit manual checking of all register argument assumptions. Normally selecting this option results in an excessive number of warnings so this flag is not selected by default in the master copy of the definition file.

The default options selected in the master copy of the definition file are meant to be used by a programmer changing code or creating new code and that wants rigorous type checking. The default options are:

- FLAG_INTEGERS, FLAG_LOCAL, FLAG_INDEXING_MISMATCHES, FLAG_UNSIGNED_BRANCHES, FLAG_LITERAL_WRITE_ADDRS, FLAG_IMPLICIT_REGISTER_USE

The only other option a programmer might want to select would be the FLAG_NODT_INDEXING. Note ASSUME_SUPPRESS is not included so that ASSUME statements don't hide some data type mismatch.

The default options selected by a person unfamiliar with this facility but interested in monitoring quality and catching gross errors would be different. In this situation, one wants to minimize the number of warnings that required involved investigation and only flag constructs that can be proofed by a person familiar with MACRO-32 and some OpenVMS conventions. In this case, one might only want to look for data type mismatches not protected by an ASSUME, literals with missing #-signs and literals which are used as the address of a write. The options for this situation would be:

- FLAG_INTEGERS, FLAG_LITERAL_WRITE_ADDRS, ASSUME_SUPPRESS

A run using these two option settings against the approximately 50, 000 MACRO-32 arguments in the OpenVMS AXP SYS facility only flagged a small percentage of arguments.

	Percentage of Arguments Flagged
• Programmer Options	9 %
• Monitoring Options	2 %

New Alpha Data Types

OpenVMS AXP has introduced new data types for integer and pointer data fields to meet the needs of Alpha development and to increase portability of OpenVMS code between Alpha and VAX platforms. These data types fall into three categories: Forced, Hardware and Software.

Forced definitions are integer or pointer fields whose data typed have been forced to be byte, word, longword or quadword under all environments, ie. to meet the requirements of a fixed public interface. The following are some of the forced data types strings:

- $IB_ = Integer Byte, $IW_ = Integer Word, $IL_ = Integer Longword, $IQ_ = Integer Quadword, $PL_ = Pointer Longword, $PQ_ = Pointer Quadword

The Forced Integer data types have been added to the definitions of the BYTE, WORD, LONG or QUAD data types. For the Forced Pointer data data types, two data types have been added PTR_L (Pointer Longword) and PTR_Q (Pointer Quadword). The Forced Pointer data types are treated like the ADDR data type and are only allowed on opcodes that treat the argument as an entire longword or quadword entity. Thus they are not used with bit manipulation or arithmetic opcodes but are used with CLR or MOV opcodes.

Hardware definitions are integer or pointer fields whose size is determined by the target hardware the code will run on. For VAX, the hardware size is 32 bits, for Alpha the hardware size is 64 bits.

Software definitions are integer or pointer fields whose size is determined by the default software environment the code will run on. For VAX, the software default is 32 bits and for Alpha Phase 1 the software default will be 32 bits but for Alpha Phase 2 and Alpha/Ultrix the software default will be 64 bits.

Thus Hardware and Software Integer and Pointer data types fluctuate depending on the target environment the code is destined for. For example, the Hardware data type on a

VAX is 32 bits and the field should be type checked as if it were a longword field (ie. MOVL is allowed but MOVQ is an error). For Alpha, the Hardware data type is 64 bits and the field should be type checked as a quadword field (ie. MOVL is an error while MOVQ is allowed). For higher-level languages keeping this field size straight is the job of the compiler but for MACRO-32 code the programmer must keep these field size distinctions straight. To assist you, the Hardware and Software data types that are type checked based on the target environment. The target environment is determined first, by whether the MACRO Compiler or VAX MACRO Assembler generated the listing file and secondly by any architectural definitions file, normally pre-pended to a Macro Compiler listing file.

Listings files generated by the VAX MACRO Assembler will assume that Hardware and Software data type size is 32 bits. Listing files generated by the MACRO Compiler will assume that the Hardware data type size is 64 bits and that the Software data type size is 32 bits.

The target environment is further qualified by any architectural definitions from the ARCH_DEFS.MAR. This file is normally pre-pended to all MACRO Compiler listing files and contains local symbols that describe the target environment:

- EVAX = 1 (Hardware data type size is 32 bits)
- VAX = 1 (Hardware data type size is 64 bits)
- ADDRESSBITS = 32 (Software data type size is 32 bits)
- ADDRESSBITS = 64 (Software data type size is 64 bits)

Once the target Hardware and Software environment is determined, the Hardware and Software data types are used in type checking whenever certain other related data types are declared legal for the various opcodes in definitions file. For example, 32-bit Hardware or Software Integer data types are legal wherever the LONG data type has been used in a DEFINE_OPCODE Statement in the definitions file. The following table shows the four data type names added for these Hardware and Software Integer and Pointer data types and the related data types that trigger the usage of Hardware and Software data types when type checking in either a 32-bit or 64-bit environment:

	32-Bit environment	64-Bit environment
	-----------------	-----------------
• INT_S - Integer Software data type	LONG	QUAD
• INT_H - Integer Hardware data type	LONG	QUAD
• PTR_S - Pointer Software data type	PTR_L	PTR_Q
• PTR_H - Pointer Hardware data type	PTR_L	PTR_Q

3 TYPCHK.COM

The Type Checker command file, TYPCHK.COM, accepts as input a VAX MACRO listing file or MACRO Compiler listing file. This command file then locates the definition file and applies the definitions to the MACRO instructions in the listing file producing a copy of the listing file with warnings embedded. The file containing the warnings has the same filename as the listing file but a file extension of ".TYPCHK" . To locate all warnings in the .TYPCHK file, search for the string that begins all warning messages: "WARN: " . All MACRO instructions even those resulting from expansions of macros are checked by the Type Checker.

The following example creates a listing file with warnings called EZDRIVER.TYPCHK from EZDRIVER.LIS.

```
$ @typchk

Enter VAX or AXP MACRO-32 Listing File to check: ezdriver

Type Checking File: DUA0:[HOULIHAN]EZDRIVER.LIS;19

$ dir *.typchk

Directory DUA0:[HOULIHAN]

EZDRIVER.TYPCHK;1

Total of 1 file.

$
```

The listing file input must be a single filespec but can contain wildcards. Wildcards would be used if one wanted to type check all the files in an entire facility for example. A default file extension of .LIS is supplied if none is specified. The output file(s) are always placed in the default directory. Note that TYPCHK.COM determines whether a VAX MACRO or MACRO Compiler listing file is input by reading the first page header line of the file and adjusts automatically to the correct format and architectural dependencies like the number of registers, (for VAX MACRO R0-R15, for Alpha R0-R31). If wildcards are used and both VAX MACRO and MACRO Compiler listing are included, then the TYPCHK command file adjusts automatically to each file’s format. If the listing file is not a VAX MACRO or MACRO Compiler listing file then an error message prints stating the file is not a Macro listing file. For wildcards, this error does not stop the wildcard searching. The error prints and the type checker goes onto the next file in the wildcard search.

TYPCHK.COM must locate the definition file and read in the definitions before type checking the listing file. Three attempts are made to locate this file:

1 The file TYPCHK.DEFINITIONS is searched for in your local directory and opened if found.

2 If the logical TYPCHK$DEFINITIONS exists then the file pointed at by this logical is opened.

3 The master copy of the definition file is opened.

If no definition file can be opened then an error is printed and the type checking is not performed.

These three mechanisms allow an engineer to override the master copy of the definition file with facility-specific definitions. By placing a facility specific definition file in a facility-specific local directory from which all type checking is done, the appropriate definition file is automatically picked up. This is useful when dealing with multiple facilities where merely switching to the correct directory picks up the appropriate definition file. The logical name allows one to point directly to the definition file without restrictions on where the default directory is currently set.

Performance of the Type Checker is a function of the size of the module. A 6000 line source module took about 1 hour in elapsed time to type check on a VAXstation 3100. This includes the 5 minutes it takes to read in all the definitions.

4 TYPCHK_DIFF.COM

One drawback of type checking is dealing efficiently with a large number of false warnings on fields that disobey conventions but have been determined to be correct. There is a tendency to define the checking rules so generally that the effectiveness of type checking is reduced for the sake of holding the number of warnings to a manageable size. TYPCHK_DIFF.COM was created to ease the comparison of two TYPCHK files with warnings embedded. You input the latest TYPCHK file and can compare against a previous TYPCHK file of a known state. Using the specialized DIFF provided by this command file, you can easily detect if new warnings have been found in a changed version of the module or if warnings for real problems have been eliminated by a change. Thus keeping previous TYPCHK files around during editing sessions becomes useful.

In the example below some changes were made to the module EZDRIVER.MAR. You will notice that multiple versions of EZDRIVER.TYPCHK exist. Before any editing of EZDRIVER took place, EZDRIVER.TYPCHK.1 was re-generated from the previous OpenVMS version of the driver which was in a known state (all of it's warnings had been verified). As each version of the driver is created, the TYPCHK file for the most recent version is checked against the previous version. In the example below, it is assumed that the TYPCHK command file was run on the latest version of the EZDRIVER to produce EZDRIVER.TYPCHK;12. The latest version, EZDRIVER.TYPCHK.12, is compared against an older version 11. Once the NEW TYPCHK file is input, the command file displays all previous versions of EZDRIVER.TYPCHK with the size and creation date/time to simply the selection of an earlier version of the TYPCHK file.

Depending on the degree of change, TYPCHK_DIFFing the final version against original version of the module, would assure that no final edits have introduced errors detectable from newly generated warnings.

```
$ @typchk_diff

DIFF New and Old .TYPCHK files to determine if warnings eliminated or introduced.


Enter NEW  .TYPCHK File : ezdriver

1) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;11    994     17-MAR-1991 11:32:14.08
2) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;10    1018    17-MAR-1991 08:50:12.87
3) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;9     1002    16-MAR-1991 16:17:43.10
4) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;8     996     16-MAR-1991 15:17:40.71
5) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;7     996     16-MAR-1991 14:17:39.19
6) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;6     995     16-MAR-1991 13:17:37.69
7) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;5     996     16-MAR-1991 12:17:36.14
8) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;4     996     16-MAR-1991 11:17:34.44
9) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;3     996     16-MAR-1991 10:17:32.48
10) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;2    996     15-MAR-1991 12:17:30.50
11) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;1    993     15-MAR-1991 11:17:26.72


Enter OLD .TYPCHK File or List Number [ DUA0:[HOULIHAN]EZDRIVER.TYPCHK;11 ]:
```

```
Comparing New File:  DUA0:[HOULIHAN]EZDRIVER.TYPCHK;12
       to Old File:  DUA0:[HOULIHAN]EZDRIVER.TYPCHK;11


WARNINGs are identified by FILENAME of filespec and the line number at end of each
warning line.


************
File DUA0:[HOULIHAN]EZDRIVER_TYPCHK_12.TYPCHK_TEMP1;1
                                                MOVL      R1,XMTPTE(R6)
WARN: XMTPTE(R6) not LONG,ADDRESS field ------------------------^           Line: 541
******
File DUA0:[HOULIHAN]EZDRIVER_TYPCHK_11.TYPCHK_TEMP1;1
************
************
File DUA0:[HOULIHAN]EZDRIVER_TYPCHK_12.TYPCHK_TEMP1;1
******
File DUA0:[HOULIHAN]EZDRIVER_TYPCHK_11.TYPCHK_TEMP1;1
                                                CMPW      #^XDFFF,VCRP$L_LAN_LENERR(R2)
WARN: VCRP$L_LAN_LENERR(R2) not WORD field ------------------------------^  Line: 5241
************

Number of difference sections found: 2
Number of different warnings found: 2
```

The inputs to the TYPCHK_DIFF comm file are straightforward. The filespec terminology used in this section is the standard OpenVMS filespec, ie. *filename.extension;version.* The first input is the most recent (or NEW) TYPCHK file that you want DIFFed. Notice in the preceding example, that the .TYPCHK file extension is provided for you if none is specified. Next, a list of previous TYPCHK versions by the same filename are displayed if any exist. You may then choose from this list by entering a listing number or selecting the default which is the previous version of the TYPCHK file. You can also input an entirely different filespec for the OLD TYPCHK file at this point if so desired. The command file then tells you which files are being compared and then prints the warning differences between the two versions.

Note that between versions 12 and 11, many lines of code where added, a simple DIFF of the TYPCHK files would have produced many differences as line numbers are all different. However the TYPCHK_DIFF command file only compares the text of the warning and the instruction pointed at by the warning (minus any text before the opcode and in any instruction comment).

TYPCHK_DIFF uses the following scratch files to accomplish it's task:

- new_typchk_file.TYPCHK_NTEMP
- new_typchk_file.TYPCHK_NTEMP1
- new_typchk_file.TYPCHK_TEMP2
- old_typchk_file.TYPCHK_OTEMP
- old_typchk_file.TYPCHK_OTEMP1

These files are all cleaned up on completion. The filenames for these scratch files, new_typchk_file and old_typchk_file, refer to the two TYPCHK files that are being compared and consist of the filename, extension and version number for the TYPCHK file as one string separated by underscores. In the preceding example, the filename of the filespec reported in the DIFF-like output identifies the TYPCHK file that contains the warning (ie. EZDRIVER_TYPCHK_12). Note that new_typchk_file and old_typchk_file can be the same if the same filename, extension and version are compared in two different directories. In this case the new_typchk_file can be identified by the file extension of TYPCHK_NTEMP1 and old_typchk_file by the file extension of TYPCHK_OTEMP1.

An examination of the DIFF-like output in the preceding example reveals that a new warning has been introduced in version 12 by an additional reference using the local symbol XMTPTE. Also a warning has been eliminated in the most recent version concerning the incorrect use of a longword field, VCRP$L_LAN_LENERR, in a compare word instruction which was replaced by the intended word field. Note that at the end of each "WARN:" line in the DIFF-like output is the line number of this warning from the TYPCHK listing file identified by the filename of the previous filespec.

If the number of differences becomes too large then you can send the warning differences to a text file by merely setting the P1 parameter to any text string on the TYPCHK_DIFF command file invocation. In the example below, P1 is set to "save" to trigger saving of the DIFFs in a file. The warning differences are saved in the current default directory in a file whose filename is the same as the filename of the NEW TYPCHK file specified and whose extension is .TYPCHK_DIFF (ie. EZDRIVER.TYPCHK_DIFF in the following example). Note that a list number is used for the OLD TYPCHK file question which in this case merely selects the default filespec.

```
$ @typchk_diff  save

DIFF New and Old .TYPCHK files to determine if warnings eliminated or introduced.


Enter NEW  .TYPCHK File : ezdriver

1) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;11    994     17-MAR-1991 11:32:14.08
2) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;10    1018    17-MAR-1991 08:50:12.87
3) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;9     1002    16-MAR-1991 16:17:43.10
4) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;8     996     16-MAR-1991 15:17:40.71
5) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;7     996     16-MAR-1991 14:17:39.19
6) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;6     995     16-MAR-1991 13:17:37.69
7) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;5     996     16-MAR-1991 12:17:36.14
8) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;4     996     16-MAR-1991 11:17:34.44
9) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;3     996     16-MAR-1991 10:17:32.48
10) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;2    996     15-MAR-1991 12:17:30.50
11) DUA0:[HOULIHAN]EZDRIVER.TYPCHK;1    993     15-MAR-1991 11:17:26.72

Enter OLD .TYPCHK File or List Number [ DUA0:[HOULIHAN]EZDRIVER.TYPCHK;11 ]: 1


Comparing New File:  DUA0:[HOULIHAN]EZDRIVER.TYPCHK;12
       to Old File:  DUA0:[HOULIHAN]EZDRIVER.TYPCHK;11


DIFFs being placed in file: EZDRIVER.TYPCHK_DIFF

SAVOIE->ty ezdriver.typchk_diff

            TYPCHK_DIFF - TYPCHK files differences command file output.


Comparing New File:  DUA0:[HOULIHAN]EZDRIVER.TYPCHK;12
       to Old File:  DUA0:[HOULIHAN]EZDRIVER.TYPCHK;11


WARNINGs are identified by FILENAME of filespec and the line number at end of each
warning line.


************
File DUA0:[HOULIHAN]EZDRIVER_TYPCHK_12.TYPCHK_TEMP1;1
                                               MOVL      R1,XMTPTE(R6)
WARN: XMTPTE(R6) not LONG,ADDRESS field ------------------------^            Line: 541
******
File DUA0:[HOULIHAN]EZDRIVER_TYPCHK_11.TYPCHK_TEMP1;1
```

```
************
************
File DUA0:[HOULIHAN]EZDRIVER_TYPCHK_12.TYPCHK_TEMP1;1
******
File DUA0:[HOULIHAN]EZDRIVER_TYPCHK_11.TYPCHK_TEMP1;1
                                               CMPW      #^XDFFF,VCRP$L_LAN_LENERR(R2)
WARN: VCRP$L_LAN_LENERR(R2) not WORD field ------------------------------^  Line: 5241
************

Number of difference sections found: 2
Number of different warnings found: 2

$
```

5 WARNINGS

This section describes all the warnings that the Type Checker can issue. The variable ARGUMENT in the messages below is the text of the argument in question from the listing file. The following warnings are produced by the Type Checker:

- ARGUMENT not used in XXXX addressing mode
  - The argument is not used in the expected addressing mode, XXXX. The only currently defined addressing mode is LITERAL. This warning results from rules supplied by the DEFINE_TYPE_MODE statement. Currently certain constant data types must be used in literal addressing mode. This warning is mainly intended to catch missing #-signs. Check that the argument in question is a valid use of a constant data type field in a non-literal context.
- ARGUMENT unexpected use of XXXX addressing mode
  - This argument is used with the unexpected addressing mode, XXXX. The only addressing mode currently defined is LITERAL. This warning is used to flag literals being used as the address to read or write information and is intended to catch situations where an address argument and a literal argument have been switched. This warning is produced by definitions established with the DEFINE_TYPE_MODE statement. Check that the argument in question is a valid use of a literal as an address argument.
- ARGUMENT is Integer not used as LITERAL
  - The argument is an integer with no #-sign or register. Integers appearing by themselves are usually errors and this is meant to catch missing #-signs or $-signs on labels (ie. where 10 should be the label 10$ in the following instruction "BBC #BDBV_DAT,BDBW_ADDR(R5),10"). This warning is controlled by the the FLAG_INTEGERS option. Check that the integer appearing by itself is intended.
- ARGUMENT using XXXX indexing on non-XXXX field
  (ie. WARN: @#^MMG$GL_SPTBASE[R0] using BYTE indexing on non-BYTE Field)
  - The data size for indexing on this argument does not match the size implied by the data type embedded in the field name. This warning is controlled by the FLAG_INDEXING_MISMATCHES option. The warning provides the type of indexing being done, XXXX, which is BYTE, WORD, LONG, QUAD or OCTA. Verify the data size of the indexing is intended for this field.
- ARGUMENT using XXXX indexing on Unknown Data Type field
  - The argument is using indexing but no embedded data type exists to check against the type of indexing being performed (ie. (R0)[R3]). This warning is controlled by the FLAG_NODT_INDEXING option. Verify that type of indexing being performed, XXXX, is intended for this argument.

- ARGUMENT not XXXX field

  — The argument is not the expected data type specified, XXXX. Note XXXX may be more than one data type. This warning is designed to catch data type mismatches. The warnings result from definitions established with the DEFINE_OPCODE statement. Note that MACRO ASSUME statements can reduce the frequency of these data type mismatch warnings, see the ASSUME_SUPPRESS option. Check that the data type mismatch of the argument is not causing unintended loss or modification of bits or unintended use or neglect of bits.

- ARGUMENT is literal used as address for write

  — Argument is a literal being used as address of a write. This warning is controlled by the FLAG_LITERAL_WRITE_ADDRS option. It is normally a bug if a literal is being used as the address of a write. Write arguments are defined in the DEFINE_OPCODE statement by "W"rite or "M"odify suffix as an argument's access type. Check that the literal is intended to be used as the address for a write.

- OPCODE instruction, R0-Rx destroyed
  (ie. WARN: MOVC5 instruction, R0-R5 destroyed)

  — This warning flags all opcodes that implicitly destroy or modify registers other the AP, FP, SP or PC. This warning is output only if the FLAG_IMPLICIT_REGISTER_USE option is selected. The OPCODE variable is the opcode involved and the destroyed registers are also named in the warning. Check that these registers are properly saved and restored over this instruction. Remember that useful information is sometimes returned in these registers or the registers may just be scratch registers and so they need not always be saved and restored.

- OPCODE1 needed after OPCODE2 operation ?
  (ie. WARN: BGTRU needed after CMPW operation ?)

  — This warning flags instances of the use of signed branch instructions after an operation on a unsigned byte or word field where an unsigned branch may be needed. This warning is only issued if the FLAG_UNSIGNED_BRANCHES option is selected. OPCODE1 is the unsigned branch opcode that is potentially needed (BGTRU, BGEQU, BLEQU or BLSSU) and OPCODE2 is the previous byte or word opcode. Using OPCODE2, look back in the listing and locate the byte or word operation that set the VAX condition code and determine if the fields being tested are unsigned. If unsigned then potentially OPCODE1 is intended.

6 ERRORS

The following errors can be produced by the TYPCHK command files:

- Error - No Definition file (TYPCHK.DEFINITIONS) is in the local directory or in the master directory (see DCL symbol MASTER_LOC in TYPCHK.COM), also logical TYPCHK$DEFINITIONS is not defined.
  - No TYPCHK Definition file can be found via the expected mechanisms. See the TYPCHK.DEFINITIONS Section which describes these mechanisms.
- Error - No such Address Mode Defined: XXXX
  - An invalid addressing mode, XXXX, was used in a DEFINE_TYPE_MODE statement in the TYPCHK.DEFINITIONS file. The defined addressing modes are specified in the TYPCHK.DEFINITIONS Section.
- Error - No such Flag: XXXX
  - An unknown option flag, XXXX, was used in a DEFINE_OPTION statement. The existing option flags are defined in the TYPCHK.DEFINITIONS Section. Correct the flag spelling and re-run the type check.
- Error - Unbalanced parentheses in a DEFINE_OPCODE statement in Definition File encountered near line: XXXX
  - The DEFINE_OPCODE statement allows multiple data types to be specified for a given argument by enclosing them in parentheses. However these parentheses cannot span multiple lines of the definition text file. If unbalanced parentheses are detected then this message will print along with XXXX which is the line or the line after the line containing the unbalanced parentheses.
- Error - Invalid Argument Suffix: XXXX
  - The DEFINE_OPCODE statement argument specified, XXXX, has an invalid suffix. The suffix can only be two characters, the first character is the access type and the second character is the data size. The comments for the DEFINE_OPCODE statement in the definition file describe the valid values for each character of the suffix.
- Error - for Opcode XXXX, Data Type YYYY is not defined.
  - The DEFINE_OPCODE statement for opcode XXXX is using an unknown data type YYYY. Define the data type YYYY in a DEFINE_TYPE statement prior to the DEFINE_OPCODE statement.
- Error - No such Data Type: XXXX
  - Data type XXXX was used in a DEFINE_TYPE_MODE statement before being defined. Define data type XXXX in a DEFINE_TYPE statement prior to the DEFINE_TYPE_MODE statement.
- Error - No such File: XXXX
  - The specified file, XXXX, does not exist or cannot be accessed by you.

- Error - Not a Macro Listing file: XXXX
  - — The specified file, XXXX, is not a VAX MACRO listing file or a MACRO Compiler listing file. This is determined by examining the first line of the listing file to see if the listing file was produced by a MACRO compiler or assembler.
- Error - No such List Number: XXXX
  - — This error is issued only by the TYPCHK_DIFF command file when an invalid list number XXXX is entered. The list number is an index into a displayed list of old TYPCHK files. Re-enter a valid list number from the list displayed.