# HARBOUR xHARBOUR DIFFERENCES

Version 0.0.2

## NOTE

This text describes most of the important differences between Harbour and xHarbour with some references to Clipper and other compatible compilers like xBase++, CLIP and FlagShip.

Many thanks to Pritpal and Viktor for updating this text. I hope that it will be updated in the future also by the xHarbour developers. It describes the status of the two compilers as at the end of October 2009:

Harbour 2.0.0 beta3 (revision 12788)

xHarbour 1.2.1 (revision 6629)

Przemek

I have formatted the text of this document for easier reading and taken the liberty of making minor changes to the phrasing of the text in places without, I hope, altering the meaning of the original authors. I have also added page numbers and a table of contents. I found the information contained in the original text file very helpful but wanted it to be easier to read and easier to find the specific information I was after at any time.

xProgrammer

# Table of Contents

MODULES....5
NEW LANGUAGE STATEMENTS....7
EXTENDED CODEBLOCKS....10
HASH ARRAYS....11
REFERENCES TO VARIABLES STORED IN ARRAYS....12
PASSING ARRAY AND HASH ITEMS BY REFERENCE....14
PASSING OBJECT VARIABLES BY REFERENCE....15
DETACHED LOCALS AND REFERENCES....16
FUNCTIONS WITH VARIABLE NUMBER OF PARAMETERS....18
MACRO MESSAGES....20
MULTIVALUE MACROS....21
USING [] OPERATOR FOR STRING ITEMS....22
NEGATIVE INDEXES IN [] OPERATOR USED TO ACCESS ITEMS FROM TAIL....23
USING ONE CHARACTER LENGTH STRING AS NUMERIC VALUE....24
OOP INTERFACE TO HASH ARRAYS....25
$ OPERATOR EXTENSIONS (arrays and hashes)....26
NEW BIT OPERATORS....27
IN, HAS, LIKE OPERATORS....28
GLOBAL / GLOBAL EXTERNAL (GLOBAL_EXTERN)....29
DATETIME/TIMESTAMP VALUES....30
LITERAL DATE AND TIMESTAMP VALUES....32
EXTENDED LITERAL STRING IN COMPILER AND MACROCOMPILER....34
SYMBOL ITEMS AND FUNCTION REFERENCES....35
OOP SCOPES....36
OOP AND MULTIINHERITANCE....37
OOP AND PRIVATE/HIDDEN DATAs....39
OOP AND CLASS OBJECT/CLASS MESSAGES....42
TYPED OBJECT VARIABLES....43
OBJECT DESTRUCTORS....44
SCALAR CLASSES....46
RUNTIME CLASS MODIFICATION....47
ARRAY AND STRING PREALLOCATION....48
DIVERT STATEMENT....49
NAMESPACEs....50
MULTI WINDOW GTs AND RUNTIME GT SWITCHING....51
MULTI THREAD SUPPORT....52
HARBOUR TASKS AND MT SUPPORT IN DOS....55
BACKGROUND TASK....56
CODEBLOCK SERIALIZATION / DESERIALIZATION....57
NATIVE RDDs....58
REGULAR EXPRESSIONS....60
INET SOCKETS....61
I18N SUPPORT....62
ZLIB....64
MACRO COMPILER....65
COMPILER LIBRARY....66
PP LIBRARY....67
LEXER....68

CONTRIB LIBRARIES....69
PORTABILITY....70
C LEVEL COMPATIBILITY....71
HBRUN / XBSCRIPT....75
HBMK2....76
PERFORMANCE AND RESOURCE USAGE....77

## COMPILE TIME SUPPORT FOR MERGING MULTIPLE .PRG MODULES

Clipper enables the compilation of many .prg modules included by @<name>.clp and/or SET PROCEDURE TO ... / DO ... [ WITH ... ] into a single output object. In such compilations it supports, separated for each .prg file, file wide declarations when the -n switch is used and allows the use of more than one static function with the same name if each of them is declared in a different .prg module. The following code illustrates such a situation:

```
/***** t1. prg *****/
static s  := "t01:s"
static s1 := "t01:s1"
proc main()
   ? "===="
   ? s, s1
   p1();p2();p3()
   ? "===="
   do t2
   ? "===="
return

proc p1        ; ? "t01:p1"
static proc p2 ; ? "t01:p2"
static proc p3 ; ? "t01:p3"
init proc pi ; ? "init t01:pi"
exit proc pe ; ? "exit t01:pe"

/***** t2. prg *****/
static s  := "t02:s"
static s2 := "t02:s2"
proc t2()
   ? s, s2
   p1();p2();p3()
return

static proc p1 ; ? "t02:p1"
proc p2        ; ? "t02:p2"
static proc p3 ; ? "t02:p3"
init proc pi ; ? "init t02:pi"
exit proc pe ; ? "exit t02:pe"
```

It needs the -n switch for file wide declarations and uses static/init/exit functions with the same names but declared in different modules. It can be compiled and linked by Clipper and Harbour as follows firstly with Clipper:

```
cl t1.prg /n/w/es2
```

or with Harbour:

```
hbmk2 t1.prg -n -w -es2
```

and then executed.

xHarbour does not have such functionality and above code has to be adapted to work with

this compiler. Additionally it does not work well with case sensitive file systems as can be seen in above example where it converts "t1" to "T1" and then tries to include "T1.prg".

For users who have old Clipper code written for DOS file systems with mixed upper and lower letters in file names used directly or indirectly by procedure name, Harbour provides compile time switches which enable automatic filename conversions for all files opened by compiler:

```
-fn[:[l|u]|-]    set filename casing (l=lower u=upper)
-fd[:[l|u]|-]    set directory casing (l=lower u=upper)
-fp[:<char>]     set path separator
-fs[-]           turn filename space trimming on or off (default)
```

This functionality is also local to Harbour so cannot be used with xHarbour as workaround for above problem though it should be easy to add it to this compiler in the future.

Both compilers support runtime switches for file name conversions.

```
SET FILECASE LOWER | UPPER | MIXED
SET DIRCASE LOWER | UPPER | MIXED
SET DIRSEPARATOR <cDirSep>
set( _SET_TRIMFILENAME, <lOnOff> )
```

which can be used in programs not intended to work with different file system(s) and with different OS(s).

# NEW LANGUAGE STATEMENTS

### 1. FOR EACH

Harbour supports all xHarbour functionality and it also offers additional features which are not available in xHarbour.

a) it enables iteration over more then one variable

```
FOR EACH a, b, c IN aVal, cVal, hVal
   ? a, b, c
NEXT
```

b) it enables iteration in a descending order through the use of the DESCEND flag, for example:

```
FOR EACH a, v IN aVal, cVal DESCEND
   ? a, b
NEXT
```

c) it has native support for hashes:

```
FOR EACH x IN { "ABC" => 123, "ASD" => 456, "ZXC" => 789 }
   ? x, "@", x:__enumKey()
NEXT
```

d) it enables iteration over strings, for example:

```
s := "abcdefghijk"
FOR EACH c IN @s
   IF c $ "aei"
      c := UPPER( c )
   ENDIF
NEXT
? s      // AbcdEfghIjk
```

e) it gives OOP interface to control enumerator variables which is very important when more then one variable is iterated or when FOR EACH is called recursively, for example:

```
hVal := { "ABC" => 123, "ASD" => 456, "ZXC" => 789 }
FOR EACH x IN hVal
   ? x:__enumIndex(), ":", x:__enumKey(), "=>", x:__enumValue(), ;
     "=>", x:__enumBase()[ x:__enumKey() ]
NEXT
```

f) it gives very flexible OOP mechanism to overload FOR EACH behavior for user defined objects adding to above enumerator methods also __enumStart(), __enumStop(), __enumSkip() methods which allow us to implement many different enumeration algorithms depending on used data

g) it does not have any hard coded limitations for recursive calls (it's limited only by available memory and HVM stack size), for example:

```
proc main()
   p( 0 )
return

proc p( n )
   local s := "a", x
   ? n
   if n < 1000
      for each x in s
         p( n + 1 )
      next
   endif
return
```

In xHarbour there is function HB_ENUMINDEX() which is supported by Harbour in XHB library.

**2. WITH OBJECT / END[WITH]**

In Harbour it does not have any hard coded limitations for recursive calls (it's limited only by available memory and HVM stack size), for example:

```
proc main()
   p( 0 )
return

proc p( n )
   ? n
   if n < 1000
      with object n
         p( n + 1 )
      end
   endif
return
```

It also uses OOP interface just like FOR EACH, so it's possible to use :__withObject() to access / assign current WITH OBJECT value.

In xHarbour there are functions HB_QWITH(), HB_WITHOBJECTCOUNTER() and HB_RESETWITH() which are supported by Harbour in XHB library.

**3. SWITCH / [ CASE / [EXIT] / ... ] OTHERWISE / END[SWITCH]**

Harbour uses a jump table with predefined values which gives a significant speed improvement in comparison to sequential PCODE evaluation (as in DO CASE statements).

In xHarbour SWITCH does not use such a jump table and generated PCODE is similar to that used for DO CASE or IF / ELSEIF. Only the main switch value calculation is optimized and reused for all statements so the speed improvement is relatively small.

xHarbour uses the DEFAULT clause instead of OTHERWISE.

Harbour supports integer numbers and strings as SWITCH values, for example:

```
switch x
         case 1          ; [...]
         case 10002      ; [...]
         case "data"     ; [...]
         otherwise       ; [...]
      endswitch
```

xHarbour only supports integer numbers and one character length strings like "A", "!", "x", " ", ...

**4. BEGIN SEQUENCE [ WITH <errBlock> ]**
**[ RECOVER [ USING <oErr> ] ]**
**[ ALWAYS ]**
**END SEQUENCE**

This is unique to Harbour. xHarbour has a limited equivalent version of these statements:

```
TRY
[ CATCH [<oErr>] ]
[ FINALLY ]
END
```

TRY gives exactly the same functionality as:

```
BEGIN SEQUENCE WITH { |e| break(e) }
```

xHarbour causes performance reduction in PCODE evaluation for the statements described above with the exception of the SWITCH statement, even if user does not use them at all. In Harbour they are implemented in different way which does not cause any overhead or performance reduction for other code.

## EXTENDED CODEBLOCKS

Both compilers support compile time extended codeblocks which allow the use of statements within such codeblocks. However the syntax used is a little different. Harbour uses the standard Clipper codeblock delimiters {}, for example:

```
? eval( { | p1, p2, p3 |
          ? p1, p2, p3
          return p1 + p2 + p3
        }, 1, 2, 3 )
```

whereas xHarbour uses <>, for example:

```
? eval( < | p1, p2, p3 |
          ? p1, p2, p3
          return p1 + p2 + p3
        >, 1, 2, 3 )
```

In Harbour extended codeblocks work like nested functions and support all function attributes. For example they can have their own static variables or other declarations which are local to the extended codeblocks and do not affect the upper function body.

The xHarbour compiler was not fully updated for such functionality and extended codeblocks were added to existing compiler structures. As a result not all language constructs work in extended codeblocks. This creates a set of very serious compiler bugs. For example the following code, which contains syntax errors, is compiled by xHarbour without a single warning message but gives unexpected results at runtime:

```
#ifndef __XHARBOUR__
   #xtranslate \<|[<x,...>]| => {|<x>|
   #xcommand > [<*x*>]       => } <x>
#endif

proc main()
   local cb, i
   for i:=1 to 5
      cb := <| p |
               ? p
               exit
               return p * 10
            >
      ?? eval( cb, i )
   next
return
```

It's possible to create many similar examples which are mostly caused by the compiler infrastructure missing nested functions support.

This could be fixed if someone invests some time to clean up the xHarbour compiler.

## HASH ARRAYS

Both compilers support for hash arrays. Hash arrays are similar to normal arrays but also allow the use of non integer values (and arbitary integer values) as indexes. They can use string, date, non integer numeric or pointer values (in Harbour) items. They can be created using list of key value pairs separated by "=>" enclosed in curly braces "{}". For example:

```
hVal := { "ABC"    => 123.45, ;
          100.1    => date(), ;
          100.2    => 10,     ;
          100      => 5,      ;
          date()-1 => .t. }
```

The items can be accessed using the [] operator, for example:

```
? hVal[ "ABC" ]      // 123.45
? hVal[ 100 ]        // 5
? hVal[ date()-1 ]   // .t.
? hVal[ 100.2 ]      // 10
? hVal[ 100.1 ]      // date()
```

By default hash items in both compilers support automatically adding new elements on an assignment operation. This can be disabled using one of the hash item functions. Harbour has an additional extension which enable automatic addition with default values for the assigned key. This also applies to the access operation and reference operator. It also supports passing hash array items by reference and some other minor extensions.

xHarbour does not support autoadd on access or reference operations and passing hash array items by reference does not work (see passing array and hash item by reference).

xHarbour has additional functionality which can be enabled for each hash array. It's an index in which information is stored about the order in which items were added to the hash array and a set of functions to operate on this index (HAA*()). They are called associative arrays in xHarbour. Harbour does not have such functionality.

Both compilers have a set of functions to perform different operations on hash arrays. They offer similar functionality. Harbour uses the HB_H prefix (for example HB_HSCAN()) whereas xHarbour uses the H prefix (for example HSCAN())

## REFERENCES TO VARIABLES STORED IN ARRAYS

In xHarbour the behavior of references stored in array is reverted in comparison to Clipper or Harbour.

In Clipper and Harbour VM executing code like:

```
aVal[ 1 ] := 100
```

clears unconditionally first item in the array aVal and assign the value 100 to it. xHarbour checks if aVal[ 1 ] is a reference and if it is resolves the reference and then assigns the value (in this case 100) to the resolved destination item.

In Clipper and Harbour VM executing code like:

```
x := aVal[ 1 ]
```

copy to x the value stored in aVal[ 1 ]. xHarbour checks is aVal[ 1 ] is a reference and if it is resolves the reference and then copies the value of resolved reference destination item to x.

It can be seen in code like:

```
proc main
   local p1 := "A", p2 := "B", p3 := "C"
   ? p1, p2, p3
   p( { @p1, p2, @p3 } )
   ? p1, p2, p3

proc p( aParams )
   local x1, x2, x3

   x1 := aParams[ 1 ]
   x2 := aParams[ 2 ]
   x3 := aParams[ 3 ]

   x1 := lower( x1 ) + "1"
   x2 := lower( x1 ) + "2"
   x3 := lower( x1 ) + "3"
```

Harbour and Clipper ouput:

```
A B C
a1 B a13
```

but xHarbour outputs:

```
A B C
A B C
```

xHarbour's behaviour is not Clipper compatible so it may cause portability problems. Code like above was used in Clipper as a workaround for the limited number of parameters (32 in Clipper) that could be passed to a function. But it allowed directly assignment to items of

arrays returned by hb_aParams() and updating corresponding variables passed by references (see functions with variable number of parameters below).

The fact that xHarbour does not have a '...' operator which can respect existing references in passed parameters and does not support named parameters in functions with variable number of parameters causes that reverted references introduce limitation, for example it's not possible to make code like:

```
func f( ... )
   local aParams := hb_aParams()
   if len( aParams ) == 1
      return f1( aParams[ 1 ] )
   elseif len( aParams ) == 2
      return f2( aParams[ 1 ], aParams[ 2 ] )
   elseif len( aParams ) >= 3
      return f3( aParams[ 1 ], aParams[ 2 ], aParams[ 3 ] )
   endif
return 0
```

which will respect references in parameters passed to f() function.

## PASSING ARRAY AND HASH ITEMS BY REFERENCE

Harbour supports passing array and hash items by reference, for example:

```
proc main()
   local aVal := { "abc", "klm", "xyz" }, ;
         hVal := { "qwe"=>"123", "asd"=>"456", "zxc"=>"789" }
   ? aVal[1], aVal[2], aVal[3], hVal["qwe"], hVal["asd"], hVal["zxc"]
   p( @aVal[2], @hVal["asd"] )
   ? aVal[1], aVal[2], aVal[3], hVal["qwe"], hVal["asd"], hVal["zxc"]

proc p( p1, p2 )
   p1 := '[1]'
   p2 := '[2]'
```

Compiled by Harbour above code shows:

```
abc klm xyz 123 456 789
abc [1] xyz 123 [2] 789
```

In xHarbour passing array items by reference works but passing hash items by reference does not work although it does not generate either a compile time or a run time error. The above code can be compiled and executed but it shows:

```
abc klm xyz 123 456 789
abc [1] xyz 123 456 789
```

## PASSING OBJECT VARIABLES BY REFERENCE

Both compilers support passing object variables by reference though this functionality in xHarbour is limited to pure instance or class variables only and does not work for SETGET methods. In Harbour it works correctly.

This code illustrates the problem:

```
proc main()
   local oBrw := tbrowseNew()
   ? oBrw:autoLite
   oBrw:autoLite := !oBrw:autoLite
   ?? "=>", oBrw:autoLite
   p( @oBrw:autoLite )
   ?? "=>", oBrw:autoLite

proc p( x )
   x := !x
```

Harbour prints:

```
.T.=> .F.=> .T.
```

but xHarbour prints:

```
.T.=> .F.=> .F.
```

without generating any compile or run time errors.

## DETACHED LOCALS AND REFERENCES

When local variables are used in codeblocks then it's possible that the codeblocks will exist after leaving the function when they were created. It's potentially very serious problem which have to be resolved to avoid internal VM structure corruption. In Clipper, Harbour and xHarbour special mechanism is used in such situation. Local variables are "detached" from VM stack so they are still accessible after leaving the function, for example:

```
proc make_cb()
   local n := 123
   return {|| ++n  }
```

We call such variables "detached locals".

Here there are two important differences between Clipper and [x]Harbour. In Clipper variables are detached when function exits (returns) and it does not know which variables were used but simply detach whole local variable frame from VM stack. It's very important to know that because it can be source of serious memory problems in OS like DOS.

This simple code illustrates it:

```
// link using RTLINK and run with //e:0 //swapk:0
// repeat test second time with additional non empty parameter <x>
#define N_LOOPS 15
#xcommand FREE MEMORY => ? 'free memory: ' + ;
                         AllTrim( Str( Memory( 104 ) ) )
proc main( x )
   local n, a
   a := array( N_LOOPS )
   FREE MEMORY
   for n := 1 to N_LOOPS
      a[n] := f( x )
      FREE MEMORY
   next
return
func f(x)
   local cb, tmp, ref
   tmp := space( 60000 )
   if empty( x )
      cb := {|| .t. }
   else
      cb := {|| ref }
   endif
return cb
```

If you execute above program with non empty parameter then 'tmp' variable is detached with codeblock which uses 'ref' variable and not released as long as codeblock is still accessible. It means that in few iterations all memory are allocated and program crashes. Clipper's programmers should know that and be careful when use detached local and if necessary clear explicitly other local variables before returning from the function by setting NIL to them.

In Harbour and xHarbour only variables explicitly used in codeblocks are detached and detaching is done when codeblock is created and original local variables are replaced by

references. It is possible because Harbour and xHarbour support multilevel references chains so it works correctly also for local parameters passed be reference from parent functions. In Clipper only one level references are supported what creates second important differences. When Clipper detaches frame with local parameters then it has to unreference all existing references breaking them. This code illustrates it:

```
proc main()
   local cb, n := 100
   mk_block( @cb, @n )
   ? "after detaching:"
   ? eval( cb ), n
return
proc mk_block( cb, n )
   n := 100
   cb := {|| ++n }
   ? "before detaching:"
   ? eval( cb ), n
return
```

Above code compiled by Clipper shows:

```
before detaching:
       101        101
after detaching:
       102        101
```

so after detaching the references to 'n' variable is broken and codeblocks access his own copy of this variables.

In Harbour it works correctly so the results are correct and it shows:

```
before detaching:
       101        101
after detaching:
       102        102
```

In xHarbour ( for unknown to me reasons ) Clipper bug is explicitly emulated though it was possible to fix it because xHarbour inherited from Harbour the same early detaching mechanism with multilevel references so just like in Clipper xHarbour programmers have to carefully watch for possibly broken references by detached locals and add workarounds for it if necessary.

## FUNCTIONS WITH VARIABLE NUMBER OF PARAMETERS

Both compilers support functions with a variable number of parameters. However in xHarbour this is limited to named parameters. It does not support unnamed parameters. In Harbour you can declare some named parameters and then unnamed ones just like in many other languages, for example:

```
func f( p1, p2, p3, ... )
```

The unnamed parameters can be used in different statements passing them by '...' operator, for example as array items:

```
proc main()
   AEval( F( "1", "2", "A", "B", "C" ), {|x, i| qout( i, x ) } )

func f( p1, p2, ... )
   ? "P1:", p1
   ? "P2:", p2
   ? "other parameters:", ...
   return { "X", ... , "Y", ... "Z" }
```

or as array indexes:

```
proc main()
   local a := { { 1, 2 }, { 3, 4 }, 5 }
   ? aget( a, 1, 2 ), aget( a, 2, 1 ), aget( a, 3 )

func aget( aVal, ... )
return aVal[ ... ]
```

or as function parameters:

```
proc main()
   info( "test1" )
   info( "test2", 10, date(), .t. )

proc info( msg, ... )
   qout( "[" + msg +"]: ", ... )
```

The '...' operator saves references when the parameters are pushed and it can be used also in codeblocks, for example:

```
bCode := { | a, b, c, ... | qout( a, b, c ), qout( "[", ..., "]" ) }
```

All parameters can be accessed also using hb_aParams() function. In xHarbour this works correctly only for functions which do not use any local parameters or have been declared with a variable number of parameters or when the number of declared parameters is not smaller than the number of passed parameters. This code illustrates it:

```
proc main()
   p1("A","B","C")
   p2("A","B","C")
   p3("A","B","C")
   p4("A","B","C")
   p5("A","B","C")
```

```
proc p1
   ? procname()+"(), parameters:", pcount()
   aeval( hb_aParams(), {|x,i| qout(i,"=>",x) } )

proc p2
   local l
   ? procname()+"(), parameters:", pcount()
   aeval( hb_aParams(), {|x,i| qout(i,"=>",x) } )

proc p3(x)
   ? procname()+"(), parameters:", pcount()
   aeval( hb_aParams(), {|x,i| qout(i,"=>",x) } )

proc p4(...)
   ? procname()+"(), parameters:", pcount()
   aeval( hb_aParams(), {|x,i| qout(i,"=>",x) } )

proc p5(a,b,c,d,e)
   ? procname()+"(), parameters:", pcount()
   aeval( hb_aParams(), {|x,i| qout(i,"=>",x) } )
```

In xHarbour it's only possible to declare all parameters as unnamed, for example:

```
func f( ... )
```

and then access them using hb_aParams() or PVALUE() (the equivalent function in Harbour is called HB_PVALUE()). There is no support for named parameters together with the … operator.

In xHarbour due to reverted behavior of references stored in array items, assigning values to items in the array returned by hb_aParams() changes the corresponding parameters passed by reference. This does not happen in Harbour where item references stored in arrays work as they do in Clipper.

## MACRO MESSAGES

Both compilers support macros as messages. Clipper does not. This example shows such macro messages usage:

```
proc main()
   memvar var
   local o := errorNew(), msg:="cargo"
   private var := "CAR"

   o:&msg := "<cargo>"
   o:&( upper( msg ) ) += "<value>"
   ? o:&var.go
```

Users who want to test it in xHarbour should change:

```
o:&( upper( msg ) ) += "<value>"
```

to:

```
o:&( upper( msg ) ) := o:&( upper( msg ) ) + "<value>"
```

because using macro messages with <op>= operators or pre / post incrementation / decrementation causes the xHarbour compiler to raise GPFs during compilation.

## MULTIVALUE MACROS

In the early days basic support for multivalue macros which can be evaluated to list of values was added to Harbour, for example:

```
? &("1,2,3")
```

should show:

```
1,  2,  3
```

The implementation of this extension was not accepted by many of the Harbour developers and it was one of the main reasons of the xHarbour fork. In Harbour it was later fully removed and implemented from scratch using different internal algorithms and structures. Now Harbour supports multivalue macros in code like:

```
proc main()
   local s1 := "'a', 'b', 'c'", s2 := "1,3", a
   ? &s1
   a := { { "|", &s1, 'x', &s2, 'y' }, 'x', &s2 }
   ? "a[1] items:"
   aeval( a[1], { |x,i| qout( i, x ) } )
   ? "a["+s2+"] =>", a[ &s2 ]
return
```

xHarbour, which inherited the original implementation, after over 6 years still cannot execute correctly above code.

## USING [] OPERATOR FOR STRING ITEMS

xHarbour supports using the [] operator to access single characters in string items. Harbour doesn't by default but it has a strong enough OOP API to allow the addition of such an extension by user at .prg level without touching the core code. It has been implemented in Harbour in xhb.lib.

This code can be compiled and executed by both compilers:

```
#ifndef __XHARBOUR__
   #include "xhb.ch" // add xHarbour emulation to Harbour
#endif

proc main()
   local s := "ABCDEFG"
   ? s, "=>", s[2], s[4], s[6]
   s[2] := lower( s[2] )
   s[4] := lower( s[4] )
   s[6] := lower( s[6] )
   ?? " =>", s
return
```

Warning!. There is one difference in above the implementation introduced intentionally to Harbour. xHarbour never generates errors for out of range indexes in the [] operator used for string items but simply returns "". For example add to the above code:

```
? ">" + s[0] + "<", ">" + s[1000] + "<"
```

If the [] operator is used for other types of items an RTE is generated. Harbour will generate an RTE in all cases. If you need strict xHarbour compatibility here then you should adopt code overloading the [] operator for strings in xhb.lib removing the RTE if this is your preference.

## NEGATIVE INDEXES IN [] OPERATOR USED TO ACCESS ITEMS FROM TAIL

xHarbour supports negative indexes in the [] operator. They are used to access items from the tail. For example aVal[ -1 ] is the same as aVal[ len( aVal ) - 1 ].

By default the Harbour core code does not give such functionality. It does, however, have a strong enough OOP API to allow the addition of such an extension by the user at .prg level without touching core code. It has been implemented in Harbour in xhb.lib.

This code can be compiled and executed by both compilers:

```
#ifndef __XHARBOUR__
   #include "xhb.ch" // add support for negative indexes in Harbour
#endif
proc main()
   local s := "ABCDEF", a := {"1", "2", "3", "4", "5", "6"}
   ? s, "=>", s[1], s[2], s[3], s[4], s[5], s[6], "=>", ;
     s[-1], s[-2], s[-3], s[-4], s[-5], s[-6]
   ? a[1], a[2], a[3], a[4], a[5], a[6], "=>", ;
     a[-1], a[-2], a[-3], a[-4], a[-5], a[-6]
return
```

Warning! see above note about indexes out of bound when used with the [] operator on string items.

## USING ONE CHARACTER LENGTH STRING AS NUMERIC VALUE

xHarbour uses one byte strings as numeric values corresponding to the ASCII value of the byte in a string, for example:

```
? "A" * 10 // 650
```

By default Harbour core code does not give such functionality. It does, however, have a strong enough OOP API to allow the addition of such an extension by the user at .prg level without touching core code. It has been implemented in Harbour in xhb.lib.

This code can be compiled and executed by both compilers:

```
#ifndef __XHARBOUR__
   #include "xhb.ch"
#endif

proc main()
   local c := "A"
   ? c * 10, c - 10, c + 10, c * " ", chr( 2 ) ^ "!"
return
```

and gives the same results.

Anyhow the emulation is not full here. It works only for .prg code. In xHarbour the standard C API functions/macros were modified to use one byte string items as numbers. This is a potential source of some very serious problems. For example ordSetFocus("1") should choose index called "1" or should it choose index 49? What value should be returned by ascan( {49,"1"}, "1" ) 1 or 2? So the Harbour developers decided to not add anything like that to core code. In Harbour functions written in C refuse to accept a one byte string as number and code like

```
? str( "0" )
```

generates runtime error instead of printing

```
'        48'
```

# OOP INTERFACE TO HASH ARRAYS

xHarbour allows access to items in a hash array using the OOP interface. hVal[ "ABC" ] := 100 can be alternatively written as hVal:ABC := 100. Using the OOP interface is slower than the [] operator but it works for all indexes which are valid upper case [x]Harbour identifiers. By default Harbour core code does not give such functionality. It does, however, have a strong enough OOP API to allow the addition of such an extension by the user at .prg level without touching core code. It has been implemented in Harbour in xhb.lib.

This code can be compiled and executed by both compilers:

```
#ifndef __XHARBOUR__
   #include "xhb.ch"
#endif

proc main()
   local hVal := {=>}
   hVal["ABC"] := 100
   hVal:QWE := 200
   hVal:ZXC := 300
   hVal:QWE += 50
   ? hVal:ABC, hVal:QWE, hVal:ZXC
   ? hVal["ABC"], hVal["QWE"], hVal["ZXC"]
return
```

Some Harbour users used to compile Harbour core code with the HB_HASH_MSG_ITEMS macro which enables such functionality directly in core code but it's not necessary with current code and it exists purely for historical reasons.

It's possible that in the future support for above macro will be removed or it will be replaced by a runtime switch which can be enabled/disabled for each hash array separately.

## $ OPERATOR EXTENSIONS (arrays and hashes)

In Harbour and xHarbour the $ operator can be used to check if some key or hash pair belongs to a hash, for example:

```
? "abc" $ { "qwe"=>100, "abc"=>200, "zxc"=>300 }
```

In xHarbour the $ operator can also be used to check if a value belongs to an array. It works like ASCAN() but with exact comparison for strings, for example:

```
? "abc" $ { "qwe", "abc", "zxc" } // result .T.
? "a" $ { "qwe", "abc", "zxc", { "a" } } // result .F.
```

By default Harbour core code does not allow to use $ operator for arrays and generates the same RTE as Clipper. It does, however, have a strong enough OOP API to allow the addition of such an extension by the user at .prg level without touching core code. It has been implemented in Harbour in xhb.lib. Both compilers can compile and execute the following code:

```
#ifndef __XHARBOUR__
   #include "xhb.ch"
#endif

proc main()
   ? "abc" $ { "qwe", "abc", "zxc" } // result .T.
   ? "a" $ { "qwe", "abc", "zxc", { "a" } } // result .F.
return
```

***Warning!*** *XBase++ also support $ operator for arrays but it makes non exact comparison so ` "a" $ { "abc" } ' gives .T. in Xbase++ and .F. in xHarbour or in Harbour when xHarbour compatibility library is used. Harbour users who need strict XBase++ compatibility should create own code to overload $ operators used for arrays which will follow exact XBase++ rules.*

*CLIP also has some code for such extension but it has two bugs.*

*1-st is a semi bug: it uses non exact comparison but reverts arguments so ` "abc" $ { "a" } ' gives .T.*

*2-nd which is critical: it has wrong stop condition so does not stop scanning when locates 1-st matching item. It should be fixed and I do not know what will be the final CLIP behavior.*

## NEW BIT OPERATORS

xHarbour support five new bit operators: &, |, ^ ^, <<, >>

```
<x> & <y>       - bit and
<x> | <y>       - bit or
<x> ^^ <y>      - bit xor
<x> << <y>      - bit shift left
<x> >> <y>      - bit shift right
```

Harbour does not have such operators but it has a set of bit functions (HB_BIT*()) which are fully optimized at compile time giving such functionality without extending the language syntax and introducing new operators:

```
<x> & <y>       => HB_BITAND( <x>, <y> )
<x> | <y>       => HB_BITOR( <x>, <y> )
<x> ^^ <y>      => HB_BITXOR( <x>, <y> )
<x> << <y>      => HB_BITSHIFT( <x>, <y> )
<x> >> <y>      => HB_BITSHIFT( <x>, -<y> )
```

## IN, HAS, LIKE OPERATORS

xHarbour has three operators defined as identifiers: IN, HAS, LIKE IN are synonyms of the $ operator and are translated by the lexer to $. They not give any new functionality. For portable code use $. HAS and LIKE are regular expressions operators. In Harbour they have the same functionality as HB_REGEXHAS() and HB_REGEXLIKE() functions:

```
<x> HAS <y>    => HB_REGEXHAS( <y>, <x> )
<x> LIKE <y>   => HB_REGEXLIKE( <y>, <x> )
```

Using identifiers as operators is not compatible with the Clipper pre processor precedence rules and introduces bugs to the language syntax. Harbour will never support them directly as operators. Using operators for regular expression introduces yet another unpleasant thing. It forces linking the regular expression library with the final application even if the application does not use RegEx at all. In Harbour the RegEx library is optional and linked only when user uses one of HB_REGX*() functions or explicitly uses the REQUEST command to include it.

## GLOBAL / GLOBAL EXTERNAL (GLOBAL_EXTERN)

xHarbour supports application wide static variables called GLOBALs. To declare a GLOBAL variable you have to put

```
GLOBAL <varname> [ := <initValue> ]
```

before the first function in compiled .prg module and use the -n compiler switch. In xHarbour GLOBAL variables cannot be declared inside a function body. If you want to use a GLOBAL variable in a different .prg module then you have to add the declaration

```
GLOBAL EXTERNAL <varname>
```

to your code before the first function in compiled .prg module and use the -n compiler switch. Also, the GLOBAL EXTERNAL declaration cannot be used in a function body. Unlike other variables GLOBALs declared in xHarbour reserve used names so they cannot be used in the same module in local function declarations, for example xHarbour cannot compile code like:

```
GLOBAL var
GLOBAL EXTERNAL var2

func F1( var )
   return var * 2

func F2()
   FIELD var2
   return var2
```

due to name conflicts. In general users should be careful using the same names for different type of variables in xHarbour.

GLOBAL variables in xHarbour need static link time bindings so they do not work with dynamically loaded PCODE functions from .hrb files or shared libraries (.dll, .so, .sl, .dyn, ...). Just like STATICs they cannot be accessed by the macro compiler.

Harbour does not support GLOBALs.

## DATETIME/TIMESTAMP VALUES

Both compilers support DATETIME/TIMESTAMP values though they use different implementation.

In Harbour it's a new type TIMESTAMP for which VALTYPE() function returns "T". It has its own HVM arithmetic similar to the one used by the DATE type but not exactly the same. The difference (-) between two TIMESTAMP values is represented as number where integer part is number of days and fractional part is time in given day. Non-exact comparison (=, >, <, >=, <=) comparison for TIMESTAMP and DATA value assumes that both values are equal if the date part is the same. Such semantics are also respected by native RDDs when mixed DATE and TIMESTAMP values are used in indexes, seeks, scopes, etc.

When a number is added to a DATE type then like in Clipper only the integer part increases (decreasea) the DATE value but when it's added to a TIMESTAMP value then the fractional part is also significant. When a TIMESTAMP value is added to a DATE value then the result is a new TIMESTAMP value. Here is some detailed information about relational and arithmetic operators with respect to TIMESTAMP values in Harbour.

Timestamp values with respect to the relational operators <, <=, >, >=, =, ==

- When two timestamp values are compared then the VM compares the date and the time parts in both values.

- When date and timestamp values are used in <, <=, >, >=, = operations then the VM only compares the date part in both values.

- When date and timestamp values are used in an == operation then VM compares the date part in both values and then checks that the time part of the timestamp is 0.

Timestamp values with respect to the mathematical operators + and -

```
<t> + <t> => <t>
<t> - <t> => <n>
<t> + <n> => <t>
<n> + <t> => <t>
<t> - <n> => <t>
<d> + <t> => <t>
<t> + <d> => <t>
<d> - <t> => <n>
<t> - <d> => <n>
```

when a number is the result or argument of a timestamp operation then its integer part is a number of days and the fractional part represents time of day.

In xHarbour the DATE type was extended to hold information about time. Clipper compatible DATE arithmetic in the HVM was modified to respect the fractional part in numbers which was used for time part.

The xHarbour DATETIME implementation introduces incompatibilities to Clipper (for

example compare Clipper and xHarbour results in code like: '? date() + 1.125' so in some cases existing Clipper code has to be carefully adapted to work correctly with xHarbour but it's fully functional solution though it needs some minor fixes in conversions between datetime values and numbers.

The difference between Harbour and xHarbour can be seen in this code:

```
proc main()
   local dVal, tVal
   dVal := date()
   tVal := dVal + {^ 02:00 }  // {^ 02:00 } timestamp
                              // constant, see below
   ? valtype(dVal), valtype(tVal)
   ? dVal; ? tVal
   dVal += 1.125  // In Clipper and Harbour it increases
                  // date value by 1
   tVal += 1.25   // it it increases timestamp value by 1 day
                  // and 6 hours
   ? dVal; ? tVal
   ? dVal = tVal  // In Harbour .T. because date part is the same
   ? date() + 0.25, date() + 0.001 == date()
return
```

Harbour shows:

```
D T
05/20/09
05/20/09 02:00:00.000
05/21/09
05/21/09 08:00:00.000
.T.
05/20/09 .T.
```

and xHarbour shows:

```
D D
05/20/09
06/25/21 02:00:00.00
05/21/09 03:00:00.00
06/26/21 08:00:00.00
.F.
05/20/09 06:00:00.00 .F.
```

Recently the “T” (TIMESTAMP) data type was introduced to xHarbour with some of the Harbour TIMESTAMP code in the RDDs but the VM was not modified to follow Harbour/RDD modifications. So now some parts of xHarbour are not synced and I cannot say what is the goal of DATETIME values and their arithmetic in the xHarbour VM for the future.

## LITERAL DATE AND TIMESTAMP VALUES

Both compilers try to support VFP like datetime constant values in the following format:

```
{ ^ [ YYYY-MM-DD [,] ] [ HH[:MM[:SS][.FFF]] [AM|PM] ] }
```

In Harbour the following characters can be used as date delimiters: "-", "/". Dot "." as date delimiter is not supported.

xHarbour supports only "/" as date delimiter.

There is no limit on number of characters in YYYY, MM, DD, HH, MM, SS, FFF parts. Only their value is important. This can be seen in the format in semi PP notation:

```
{ ^ <YEAR> <sep:/-> <MONTH> <sep:/-> <DAY> [[<sep2:,>]
  [ <HOUR> [ : <MIN> [ : <SEC> [ . <FRAQ> ] ] ] [AM|PP] ] }
```

NOTE: there is one important difference between Harbour and VFP/xHarbour in decoding the above format. In VFP and xHarbour when date part is missed then it's set by default to: 1899-12-30 so this code:

```
{ ^ 12:00 }
```

gives the same results as:

```
{ ^ 1899/12/30 12:00 }
```

Harbour does not set any default date value when a timestamp constant contains only time part.

Harbour supports VFP syntax only in it's compiler. It's not supported in it's macro compiler and it can be disabled in the future so it's not suggested to use it with Harbour programs.

Both compilers also supports date constants in the form 0dYYYYMMDD for example:

```
0d20090520.
```

It's also supported by the macro compilers.

Only Harbour supports date constants (in compiler and macro compiler) in the form d"YYYY-MM-DD" for example:

```
d"2009-05-20"
```

Also delimiter "/" or "." can be used instead of "-".

Only Harbour supports timestamp constant (in compiler and macro compiler) in the form "YYYY-MM-DD HH:MM:SS.fff"

The exact accepted timestamp pattern is is:

```
YYYY-MM-DD [H[H][:M[M][:S[S][.f[f[f[f]]]]]]] [PM|AM]
```

for example:

```
tValue := t"2009-03-21 5:31:45.437 PM"
```

or:

```
YYYY-MM-DDT[H[H][:M[M][:S[S][.f[f[f[f]]]]]]] [PM|AM]
```

with literal "T" as date and time part delimiters (XML timestamp format), for example:

```
tValue := t"2009-03-21T17:31:45.437"
```

The following characters can be used as date delimiters: "-", "/", "." if PM or AM is used HH is in range < 1 : 12 > otherwise in range < 0 : 23 >

## EXTENDED LITERAL STRING IN COMPILER AND MACROCOMPILER

Harbour and xHarbour support extended strings with C like escaped values as e"...", for example:

```
? e"Helow\r\nWorld \x21\041\x21\000abcdefgh"
```

They works in compiler and macro compiler but the xHarbour macrocompiler does not support strings with an embedded 0 so they are not fully functional there.

## SYMBOL ITEMS AND FUNCTION REFERENCES

Harbour supports SYMBOL items ( VALTYPE(funcSym) == "S" ) which can be used as function or message references. They have similar functionality to SYMBOL objects in Class(y) and understand NAME, EXEC and EVAL messages.

They can be created literally by the compiler using @<funcName>(), f.e:

```
funcSym := @str()
```

and in such cases they also create an explicit reference (link time binding) to given functions or by the macro compiler, for example:

```
funcSym := &("@upper()")
```

what does not create any explicit bindings. This is simple code which uses symbol items:

```
proc main()
   local funcSym
   funcSym := @str()
   ? funcSym:name, "=>", funcSym:exec( 123.456, 10, 5 )
   funcSym := &("@upper()")
   ? funcSym:name, "=>", funcSym:exec( "Harbour" )
return
```

xHarbour does not have such functionality though it can create at compile time function references using a somewhat different syntax:

```
( @<funcName>([<anyExpList,...>]) )
```

the different syntax is the side effect of bugs in the grammar rules only and will probably be fixed somewhere (macro compiler does not support such syntax at all).

In xHarbour the above code creates a pointer item ( VALTYPE(funcSym) == "P" ) which can be used in some cases as in Harbour but because the xHarbour VM does not know if a given pointer item is a function reference or not then in such a context xHarbour has to accept any pointer items as function references so any user mistake can cause a GPF or some HVM structure corruption.

## OOP SCOPES

Harbour and xHarbour both support scopes in class declarations.

It's possible to declare exported, protected and hidden messages and then at runtime scopes are checked. There are important differences in both implementations. In xHarbour for internal implementation .prg module symbol table was used so all classes declared in the same .prg file (or #included from the same .prg file) and also all other functions in this file have exactly the same scope and works like internal methods which can access hidden data. Harbour is Class(y) compatible so each class and external functions are separated. If the programmer needs to give some rights to other classes or external functions then he can explicitly declare friend classes and functions as follows:

```
FRIEND CLASS <ClassName,...>
```

and:

```
FRIEND FUNCTION <FuncName,...>
```

It's also possible to give unprotected access to all other functions declared in the same .prg module just like in xHarbour by using the MODULE FRIENDLY clause in aclass declaration, for example:

```
CLASS myCls FROM parent MODULE FRIENDLY
   [...]
ENDCLASS
```

So in Harbour all scopes aspects are explicitly controlled by programmer and AFAIK there are no side effects forced by internal implementation.  Important is also the fact in Harbour each codeblock block created explicitly or by macrocompiler inherits rights from functions when it was created and then has the same scope. It means that it's not possible to break scope protection using codeblocks and it's very important functionality for real private (hidden) data because it makes it accessible for codeblocks created by object methods.

AFAIK xHarbour does not have such functionality and codeblock scopes depends on module symbol table.

## OOP AND MULTIINHERITANCE

Harbour and xHarbour support multiple inheritance just like Class(y). Anyhow only Harbour correctly resolves possible name conflicts by casting. xHarbour and Class(y) in Clipper does not work correctly if few ancestors have instance variables with the same name even if explicit casting is used. This code illustrate the problem:

```
#ifdef __HARBOUR__
   #include "hbclass.ch"
#else
   #include "class(y).ch"
#endif

proc main()
   local o
   o := mycls():new()
   o:var := "VAR"
   o:cls1:var := "VAR1"
   o:cls2:var := "VAR2"
   o:cls3:var := "VAR3"
   ? o:var, o:cls1:var, o:cls2:var, o:cls3:var
   o:var := "[var]"
   o:cls1:var := "[var1]"
   o:cls2:var := "[var2]"
   o:cls3:var := "[var3]"
   ? o:var, o:cls1:var, o:cls2:var, o:cls3:var
return

CREATE CLASS MYCLS FROM CLS1, CLS2, CLS3
EXPORTED:
   VAR      VAR
ENDCLASS

CREATE CLASS CLS1
EXPORTED:
   VAR      VAR
ENDCLASS

CREATE CLASS CLS2
EXPORTED:
   VAR      VAR
ENDCLASS

CREATE CLASS CLS3
EXPORTED:
   VAR      VAR
ENDCLASS
```

Expected results are:

```
VAR VAR1 VAR2 VAR3
[var] [var1] [var2] [var3]
```

and Harbour shows such results.

xHarbour and Class(y) gives wrong results:

```
VAR VAR3 VAR3 VAR3
```

```
[var] [var3] [var3] [var3]
```

Please note that Harbour does not make any tricks with compile time static bindings. It's uses only dynamic bindings and resolves all problems with instance area super casting.

Correct instance area super casting is very important functionality when program is created by few programmers and it's necessary to create classes which inherits from many other ones written be different programmers to resolve possible name conflicts. So for bigger projects is one of the most important thing.

## OOP AND PRIVATE/HIDDEN DATAs

In bigger projects where different programmers works on different classes which are later used as ancestors of some new classes it's extremely important to give support for real private (hidden) data for each class programmer which will not cause problems if name conflicts appears. Otherwise each method and instance variable name used by each programmer has to be agreed with project manager to eliminate possible name conflicts. In really big projects such name conflicts can be nightmare which eliminates some languages.

Harbour resolves this problem automatically. All hidden variables and messages are automatically casted to the class which is the owner of calling code. To make it working Harbour needs fully functional scope protection and multiinheritance (see OOP SCOPES and OOP AND MULTIINHERITANCE). Everything is done automatically without any explicit casting or other then declaring variables or methods as HIDDEN source code modification. This code illustrates it. We have two classes and each of them has its own _private_ 'temp' instance variable and 'set' method. 3-rd class inherits from both of them and executes it's public methods which internally access hidden ones but without any explicit casting. In Harbour all is done automatically by HVM:

```
#include "hbclass.ch"
PROC MAIN()
   LOCAL o
   o := mycls():new()
   o:action( "X" )
   o:action( "Y" )
   WAIT
RETURN

CREATE CLASS MYCLS FROM CLS1, CLS2
EXPORTED:
   METHOD   ACTION
ENDCLASS

METHOD ACTION( x )
   ? ::plus( x )
   ? ::minus( x )
RETURN Self


CREATE CLASS CLS1
HIDDEN:
   VAR      TEMP
   METHOD   SET
EXPORTED:
   METHOD   PLUS
ENDCLASS

METHOD SET( x )
   IF ::TEMP == NIL
      ::TEMP := "C1"
   ENDIF
   ::TEMP += ":" + UPPER( x )
RETURN Self

METHOD PLUS( x )
```

```
RETURN ::SET( x ):TEMP


CREATE CLASS CLS2
HIDDEN:
   VAR      TEMP
   METHOD   SET
EXPORTED:
   METHOD   MINUS
ENDCLASS

METHOD SET( x )
   IF ::TEMP == NIL
      ::TEMP := "c2"
   ENDIF
   ::TEMP -= ">" + LOWER( x )
RETURN Self

METHOD MINUS( x )
RETURN ::SET( x ):TEMP
```

Expected results from this code are:

```
C1:X
c2>x
C1:X:Y
c2>x>y
```

and Harbour shows exactly such results.

Neither xHarbour nor Class(y) has such functionality what seems to be very serious limitation for big projects.

In xHarbour above code shows:

```
c2>x
c2>x>x
c2>x>x>y
c2>x>x>y>y
```

To compile it with classy it's necessary to make some minor modification and change in CLS1:

```
METHOD SET
```

to:

```
MESSAGE SET METHOD SET1
```

and in CLS2

```
METHOD SET
```

to:

```
MESSAGE SET METHOD SET2
```

updating method names in the code.

Then it can be compiled but like in xHarbour only one SET method is visible and only one TEMP instance variable so it shows only:

```
C1:X
```

and then generates scope violation error because unlike xHarbour Class(y) does not define all classes as MODULE FRIENDLY.

## OOP AND CLASS OBJECT/CLASS MESSAGES

Neither Harbour nor xHarbour supports class objects, class messages and class instance variables like Class(y) does.

But both compilers support shared variables which can be shared with all object of the same class or shared variables which can be shared with all object of given class and its descendants. Class methods are not supported at all by both compilers (they cannot be without class object) though xHarbour wrongly use class messages as normal messages.

In the future we plan to introduce real class objects like in Class(y) and some other languages (i.e. XBASE++) so it's important for portability to write code which is Class(y) friendly and never use class function directly as constructor, i.e. instead of writing code like:

```
o := mycls( p1, p2, p3 )
```

programmer should use:

```
o := mycls():new( p1, p2, p3 )
```

otherwise the code will not work with future Harbour versions supporting real class objects.

## TYPED OBJECT VARIABLES

Harbour support strong typed object variables, for example:

```
CREATE CLASS MyClass
   VAR   var1 AS INTVAR
   VAR   var2 AS NUMERIC
   VAR   var3 AS DATE
   VAR   var3 AS CHARACTER
ENDCLASS
```

And validates assigned values at runtime just like in Class(y). xHarbour can compile above code but AS <type> is only source code decoration for this language.

## OBJECT DESTRUCTORS

Both Harbour and xHarbour support object destructors and both compilers uses reference counters and garbage collector to execute destructors. Anyhow the low level implementation is different in the two compilers. In Harbour HVM is reentrant safe so the implementation of destructors is much simpler. It also keeps full control about reference counters and detects user code errors bound with complex items in .prg and C code what greatly helps to detect problems in user code or 3-rd party libraries and gives protection against internal HVM memory corruption. In the last years it also helped to located few very hard to exploit bugs in core code.

In xHarbour there is some basic protection but it was never fully functional. Sometimes it may cause that internal error message is generated but in most of cases internal HVM memory is silently corrupted. A good example which illustrates what may happen is in Harbour SVN in harbour/tests/destruct.prg. This code can be compiled and executed by Harbour and xHarbour. But only Harbour version detects user errors generating RTE and is necessary in all cases to keep internal memory cleaned protecting against corruption. The xHarbour version behavior is random because this .prg code corrupts internal HVM memory. It's possible that it will be executed without any visible errors or it will generate GPF or internal error. But in all cases it can be well seen using tools like valgrind or CodeGuard that it corrupts internal memory so the results are unpredictable. I.e. this is part of valgrind log generated during execution of above destruct.prg compiled by xHarbour:

```
==22709== Invalid read of size 2
==22709==    at 0x426E235: hb_gcItemRef (garbage.c:646)
==22709==    by 0x425EDE5: hb_memvarsIsMemvarRef (memvars.c:2396)
==22709==    by 0x426E674: hb_gcCollectAll (garbage.c:847)
==22709==    by 0x426E899: HB_FUN_HB_GCALL (garbage.c:1179)
==22709==  Address 0x485b096 is 22 bytes inside a block of size 56 free'd
==22709==    at 0x4023B7A: free ()
==22709==    by 0x42837A9: hb_arrayReleaseBase (arrays.c:1588)
==22709==    by 0x428381A: hb_arrayRelease (arrays.c:1602)
==22709==    by 0x4256B94: hb_itemClear (fastitem.c:248)
[...]
==22709== Invalid write of size 2
==22709==    at 0x426E4C6: hb_gcItemRef (garbage.c:652)
==22709==    by 0x425EDE5: hb_memvarsIsMemvarRef (memvars.c:2396)
==22709==    by 0x426E674: hb_gcCollectAll (garbage.c:847)
==22709==    by 0x426E899: HB_FUN_HB_GCALL (garbage.c:1179)
==22709==  Address 0x485b096 is 22 bytes inside a block of size 56 free'd
==22709==    at 0x4023B7A: free ()
==22709==    by 0x42837A9: hb_arrayReleaseBase (arrays.c:1588)
==22709==    by 0x428381A: hb_arrayRelease (arrays.c:1602)
==22709==    by 0x4256B94: hb_itemClear (fastitem.c:248)
[...]
==22709== Invalid read of size 4
==22709==    at 0x426E223: hb_gcItemRef (garbage.c:603)
==22709==    by 0x425EDE5: hb_memvarsIsMemvarRef (memvars.c:2396)
==22709==    by 0x426E674: hb_gcCollectAll (garbage.c:847)
==22709==    by 0x426E899: HB_FUN_HB_GCALL (garbage.c:1179)
==22709==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==22709==  Process terminating with default action of signal 11 (SIGSEGV)
==22709==  Access not within mapped region at address 0x0
```

Harbour executes above code cleanly without any errors:

```
==28376== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 3 from 1)
==28376== malloc/free: in use at exit: 0 bytes in 0 blocks.
==28376== malloc/free: 6,164 allocs, 6,164 frees, 10,323,064 bytes allocated.
==28376== All heap blocks were freed -- no leaks are possible.
```

Missing protection and error detection is not only problem for programmers. It also causes that in xHarbour core code few bad bugs have not been fixed so far though they exist for years. It also extremely time consuming problem for core developers when users or 3-rd party developers reports problems with some memory corruption and it's necessary to locate the reason. I remember how much time I lost for such things working on xHarbour in the past so later when I moved to Harbour it was one of the main goal for me to resolve the problem.

In xHarbour there is disabled code which was designed to detect such problems and it can be enabled by setting HB_ARRAY_USE_COUNTER_OFF macro but it causes horrible runtime overhead and was never finished to be ready as production extension. Such solution seems to be dummy way so I do not believe that it will be ever finished.

## SCALAR CLASSES

Both compilers support scalar classes and allows to add OOP functionality to native types like numeric, character, array, hash, codeblock, date, … It's possible to overload default scalar classes provided by Harbour and xHarbour or use ASSOCIATE CLASS command to bound any class with some native type. It's also possible to overload the behavior of some operators if it's not already defined for given types. Anyhow it's not possible to change operator precedence which is the same for all types and defined at compile time.

## RUNTIME CLASS MODIFICATION

Harbour and xHarbour use dynamic bindings so it's possible to modify class definitions at runtime. To avoid possible very serious problems which can be caused by such class modification Harbour provides LOCKED clause in class definition. xHarbour does not have such functionality and even documented command like OVERRIDE CLASS or EXTEND CLASS as official features though they break inheritance scheme and never worked correctly in the whole xHarbour history. They are available in Harbour only in xHarbour compatible library after including xhbcls.ch for porting existing xHarbour code to Harbour and work exactly like in xHarbour having the same problems so they are not planned to be part of Harbour core code.

## ARRAY AND STRING PREALLOCATION

Both compilers uses array and string preallocation to optimize resize operation, for example in code like:

```
s := ""
for i := 0 to 255
   s += chr( i )
next
```

or:

```
a := {}
for i := 0 to 255
   aadd( a, repl( chr( i ), 5 ) )
next
```

the string inside 's' variable and array inside 'a' variable are not resized 256 times but much seldom. It gives noticeable speed improvement with the cost of additional memory used by application. Anyhow in Harbour internal HVM and item structures are smaller then in xHarbour so usually the total memory usage in Harbour, even with extensive preallocation, is smaller then in the same application compiled by xHarbour with disabled string and array preallocation.

The speed difference can be really huge in some application. It's possible to create test code which will be 100 times faster due to this feature so it's really important functionality.

The internal implementation of these optimization in both compilers is different. In xHarbour only chosen left side expression are optimized (in practice only 'var += ...' and 'arr[ <n> ] += ...') but in Harbour all with the exception to 'o:<msg> += ...' optimization which can be enabled only optionally at Harbour compile time using HB_USE_OBJMSG_REF macro (there is no -k? option to control it by user yet), it can be well seen in code like:

```
proc main()
   memvar v
   local t, i, s
   private v := ""
   t := secondsCPU()
   s := "v"
   for i := 1 to 100000
      &s += l2bin( i )
   next
   t := secondsCPU() - t
   ? "time:", t, "sec."
return
```

it's enough to compare time difference:

```
Harbour  0.13 sec.
xHarbour 15.91 sec.
```

## DIVERT STATEMENT

xHarbour supports

```
DIVERT TO <pFuncPointer> | <pMessagePointer> ;
       [ OF <oNewSelfOrNil> ] [ FLAGS <nDivertFlags> ]
```

statement which allow to switch current function/method execution context to the new one. The detail description of the above extension can be found in xHarbour ChangeLog.027 at:

```
2007-12-07 22:30 UTC-0500 Ron Pinkas <ron/at/xharbour.com>
```

It's very dangerous extension which needs really deep knowledge about compiler and HVM internals to use it safely, because, for called function HVM does not create new stack frame. Instead it simply adopts current one. It means that programmer using above statement have to exactly know HVM registers used by different language constructions and which ones are correctly saved and then restored or cleaned by DIVERT statement. Any mistake can cause unpredictable behavior like GPF or internal HVM structure corruption.

The worst thing in above extension is the fact that it effectively blocks using HVM stack by compiler for optimization (i.e. to store some values in temporary stack variables) and also adding some new language constructions which may try to use HVM stack as temporary holder for some values so in the future such new language constructions will have to use explicitly defined new HVM registers just like the ones defined in current xHarbour VM for WITH OBJECT or FOR EACH statements increasing unnecessary overhead also in code which will not use either DIVERT TO or new language constructions, so it introduces very serious problem for further compiler improvement and syntax extensions.

In theory it should be faster method to switch the execution context than normal call but in practice it introduces new overheads directly or indirectly (by blocking some possible improvements) so in summary it causes slowness.

Probably only some examples which uses DIVERT TO with bigger number of parameters in a loop can show some speed improvement but in real life application in most of cases it's necessary to call functions/ procedures/methods in standard way which will be slower so in fact it reduces overall performance.

I tried to make some tests with above statement but quite fast I exploited few bugs in the pure DIVERT TO implementation in HVM so I was not able to make serious tests for interactions with different language constructions. xHarbour simply GPFs or makes some other strange things before and I do not have time to check what exactly is broken in this case or analyze possible bad side effects looking at xHarbour core code. I only scanned xHarbour core code for real life DIVERT TO usage and the only one practical usage of above extension in xHarbour core code is workaround for missing '...' operator working like in Harbour (see: FUNCTIONS WITH VARIABLE NUMBER OF PARAMETERS) so I do not think that Harbour users will ever ask about it.

For all of the above reasons I think that Harbour will never have anything like that at least as documented language extension.

## NAMESPACEs

xHarbour has compile time support for namespaces which uses by default static bindings with optional support for dynamic bindings in HVM which is necessary for .hrb and macrocompiler. To exchange information about defined by user namespaces xHarbour generates <namespace>.xns files during compilation of .prg ones which define public functions in given namespace. These files are automatically included by xHarbour compiler if namespace is requested in PRG code. They are also necessary for C compiler to resolve static bindings so they are explicitly included in .c files generated by xHarbour.

xHarbour compiler seems to always overload existing .xns files and there is no description for format of .xns files. It means that without some manual modification namespace can be defined only in single .prg file.

In xharbour CVS in doc/namespace.txt is basic description for used syntax but I cannot find any deeper information about expected functionality so it's hard for me to comment on it. I simply do not know the final goals for this feature and why it was introduced. I do not even know if it has been finished or not and some modifications/extensions are planned. After few tests I can only say that it does not work correctly on case sensitive file systems so probably it's not widely used functionality if no one reported the problem so it has not been fixed. If xHarbour developers can make some short description for this feature then I'll add it to this text.

Harbour does not have such functionality and AFAIK none of developers has planed to add similar to xHarbour namespace functionality. Also none of Harbour users requested about it so seems that it will not be added to Harbour.

Anyhow Harbour users and developers requested about functionality which allow to control public symbol scopes at runtime. It's necessary to create limited in functionality (reduced list of public functions which can be accessed) environment to execute untrusted code, i.e. loaded from .hrb files or dynamically compiled and registered .prg code.

In the future I plan to work on such functionality. If possible I'll try to resolve also some other problems which can appear in programs loading and executing external code like support for thread local public symbols. It should greatly help to improve some programs like HTTP servers. Probably this code should also help in implementing dynamic namespace support so maybe we introduce it but seems that it will be completely different thing then in current xHarbour.

## MULTI WINDOW GTs AND RUNTIME GT SWITCHING

Harbour allows to initialize more then one GT drivers at runtime and switch between active GTs in single application. It can be done using the following functions:

```
HB_GTCREATE( <cGtName>, [<nStdIn>], [<nStdOut>], [<nStdErr>] ) -> <pGT>
HB_GTSELECT( <pGT> ) -> <pPrevGT>
```

if all references to allocated GT drivers are cleaned in visible items then GT windows is automatically closed.

Number of allocated GTs depends on existing resources. I.e. some GTs like GTWVT or GTXWC for each hb_gtCreate() call creates new console window so application can create many different windows and switch between them. It's very nice feature for MT mode because current GT driver in Harbour is thread attribute so each thread in Harbour can allocate it's own independent console window if platform support GT driver with such possibility (graphical GTs like GTWVT, GTXWC, GTWVG). Some GTs like GTWIN, GTOS2 or GTDOS have to share the same resources (OS console windows, video screen buffer) so their parallel usage is limited but allocating them does not block allocating any other GTs, i.e. application can use one GTWIN console (MS-Windows console window) and many GTWVT console windows.

This feature can be used also with non graphical GTs which uses stream input/output like GTTRM, GTCGI or GTSTD, i.e. user can create terminal server and for each internet connection allocate new GTTRM or GTSTD driver or can execute CGI procedures in newly allocated GTCGI drivers with direct output redirection to given handle passed to hb_gtCreate(). In some other cases it can be usable to execute some part of code using GTNUL driver to pacify all output and disable input.

This functionality is also supported by virtual GT drivers GTCTW used to emulate Clipper Tools window system. GTCTW has also separate support for MT mode and keep current window as thread attribute so each thread can use its own CT window independently.

In Harbour contrib there are some GTs and mixed GT/GUI libraries which extensively use multi window functionality like GTWVG, GTQTC.

Not all GTs in Harbour has been adopted to work simultaneously in multi thread mode with more then one instance so user should not allocate more then one instance of the following GTs GTDOS, GTWIN, GTOS2, GTCRS, GTSLN, GTPCA, GTALLEG.

These GTs are limited by resources or have alternative GT drivers with similar functionality so it's hard to say if Harbour developers invest time in updating them.

The above functionality is local to Harbour and does not exists in xHarbour. Anyhow xHarbour borrowed from Harbour most of new GT code so it should be possible to implement (at least partially due to limited in xHarbour MT mode) some of the above features.

## MULTI THREAD SUPPORT

Both compilers support multi thread programming and in both compilers to use threads it's necessary to use MT version of VM.

In Harbour it's only single HVM library because in both versions (ST and MT) Harbour VM gives exactly the same public API so it's not necessary to create separate versions of any other libraries or create different .prg code for MT and ST modes. The separate ST version of VM can be linked with applications which do not create any threads to improve the performance because it's a little bit faster then MT VM. The speed difference can be compared using speedtst.prg (results below). For other platforms/C compilers it may be a little bit bigger or smaller anyhow it's not very huge and Harbour in MT mode is still much faster then xHarbour in ST mode.

For .prg programmer the only difference between applications linked with single thread VM and multi thread VM is in hb_threadStart() results. This function linked with ST VM always fails and new thread is never created.

Harbour gives set of functions with hb_thread*() and hb_mutex*() prefix to manage threads and synchronize them. It also supports THREAD STATIC variables what allows to easy adopt any existing code to MT mode. Whole Clipper compatible .prg API using internally static variables like GET system was adopted for MT mode and work with thread local statics.

Most of core code had been rewritten to be reentrant safe so it does not have to be protected by any mutexes what greatly helps in scalability. Below is a short description which shows which resources are thread local which ones are shared and how they are initialized.

**Thread attributes inherited from parent thread:**

- code page
- language
- SETs
- default RDD
- active GT driver/console window
- I18N translation set

**Thread attributes initialized to default value:**

- public variable GetList := {} (of course if public variables are not shared with parent thread).
- error block (initialized by calling ERRORSYS())
- math error handler and math error block
- default macro compiler features setting (controlled by hb_setMacro())
- RDDI_* settings in core RDDs (some 3-rd party RDDs may use global settings)
- thread static variables

**Resources which can be shared or copied form parent to child thread on user request when thread is created:**

- public variables
- private variables

**Global resources synced automatically:**
- functions and procedures
- class definitions
- RDDs
- GT drivers
- lang modules
- codepage modules

**Global resources:**
- .prg static variables (Harbour core code does not use them)

**Thread local resources:**
- memvar (public and private) variables (except the ones which are shared with parent threads)
- work areas
- thread static variables

In practice the only one not synced automatically element in Harbour is write access to the same item with complex variable. This part have to be synced by user and automatic synchronization here will strongly reduce scalability.

Harbour also support xBase++ compatible MT API with SYNC and CLASS SYNC methods, SIGNAL and THREAD classes, workarea zones and thread functions. It allows to easy port xbase++ code to Harbour. The main difference between Harbour and xbase++ is write protection to complex items. Harbour gives only read protection so threads can access the same complex variable simultaneously without any restrictions as long as it's not modified (assigned) by some other thread. Assign operations to variable which can be accessed simultaneously by other threads need explicit synchronization by user code. In xBase++ it should be synced automatically so users have to use own synchronization only to keep synced his application logic.

For sure xBase++ is more user friendly here. Anyhow such solution strongly reduce scalability so the cost of such automatic protection is very high. Additionally in most of cases simultaneous access and assign operations on the same complex variable need additional user synchronization to eliminate race condition so it does not help much in real programming. Anyhow it's interesting in some cases so it's possible that we add such functionality to Harbour in alternative HVM library to not reduce current performance.

In xHarbour for MT mode it's necessary to use different set of RTL libraries not only MT version of VM lib. MT mode change the behavior of core features and many parts of xHarbour core code have been never adopted to MT mode so it's hard to clearly define which resources are thread local and which ones are global and what should be synchronized by user. Many things are not synced at all and race conditions can cause internal HVM structure corruptions, i.e. class creation code where many threads can try to register the same class simultaneously so it needs yet a lot of work to reach real production

functionality. It also needs serious cleanup because there are race conditions even in basic MT API. Missing from the beginning clear definition of MT mode caused that in last years not all xHarbour modifications were MT safe and some of them even broke existing functionality. xHarbour MT API is present only in MT version of HVM and supports only some subset of Harbour functionality.

The only one xHarbour MT feature not existing in current Harbour code is support for:

```
CRITICAL [STATIC] FUNCTION | PROCEDURE( [<xParams,...> )
```

but in current xHarbour CVS such functionality is broken. The problem can be seen at compile time so looks that none of core developers were using it in last years. This feature in xHarbour was implemented only in .c code generated by xHarbour compiler (-gc[0-3] output) so it was never working with .hrb (-gh) files or some interpreters like xBaseScript.
xHarbour has support for SYNC methods which were designed to replicate xBase++ functionality but their real behavior is not xBase++ and Harbour compatible at all. It seems to be close to SYNC CLASS method in xBase++ and Harbour though it's not exactly the same. There is no support for real SYNC methods like in xBase++ and xBase++ MT programming functions and classes.

In summary MT mode in xHarbour looks like a work in progress, started few years ago and never finished. Instead some other core code modifications in last years systematically introduced new non MT safe code and in few cases even broke existing MT extensions.

I cannot say if xHarbour developers plan to make sth with it. Now I cannot even describe it well because looking at the xHarbour source I do not know what is intentional decision, what is a bug in implementation and what is unfinished code, i.e. such code:

```
JoinThread( StartThread( @func() ) )
```

in xHarbour has race condition and can generate RT error if new thread finished before join operation begin. For me it's bad design decision or fatal side effect of wrong implementation but I do not plan to guess. This is very trivial example and I see in core code much more serious and more complicated problems I had to resolve in Harbour which are not touched at all in xHarbour.

I know that some people created MT programs using xHarbour but for me current xHarbour MT mode is not production ready. At least it is very far from current Harbour functionality and quality.

# HARBOUR TASKS AND MT SUPPORT IN DOS

Harbour supports threads also in system without native thread support. It has special code which emulates task switching so Harbour MT programs can be compiled and executed in systems like DOS.

Harbour tasks are used by default in DOS builds in MT HVM. They can be enabled in all other builds instead of native thread support though on some platforms not tested so far which does not support POSIX <ucontext.h> task code may need some modifications to safe/restore execution context.

For .prg code Harbour tasks behave like native threads but it's important to know that Harbour does not add in some magic way MT support to OS-es like DOS or to CRTL. If one task executes OS or CRTL code then the context is not switched. It will be switched when it will return from such code.

## BACKGROUND TASK

In xHarbour after setting _SET_BACKGROUNDTASKS to .T. main HVM loop periodically (once per some number of iterations defined by user in _SET_BACKGROUNDTICK) executes user defined code interrupting current context. This functionality is called Background Tasks. Because it's defined in main HVM loop then it does not work for code generated with -gc3 compiler switch. This functionality is also enabled for MT HVM version though with different internal semantic. When _SET_BACKGROUNDTASKS is .T. all threads try to execute user defined code using thread local HVM loop counter initialized to some fixed value (HB_VM_UNLOCK_PERIOD=5000) instead of_SET_BACKGROUNDTICK.

Harbour does not have such functionality and probably will not have in such form for the following reasons:

1. it works only from HVM loop so it will not work for code which does not use it, for example generated with -gc3 compiler switch

2. such implementation causes same speed overhead even if programmer does not use background tasks

3. the frequency of executing background task is very irregular

4. in MT programs background tasks are executed in different way then in ST mode and the frequency depends on number of threads and their activity.

5. in Harbour MT programs such functionality can be very easy implemented by user without touching core code.

6. it's possible that in the future HVM will support asynchronous events what is more general mechanism and also allows to easy implement background tasks without touching core code.

## CODEBLOCK SERIALIZATION / DESERIALIZATION

xHarbour has support for codeblock serialization. It's done by simple storing compiled codeblock body (PCODE) in string variable and then restoring it from such variable. Harbour does not support it intentionally and in such form it will never be supported by HVM because it can work only in some very limited situations and then can be source of very bad bugs like internal HVM memory corruption or absolutely unexpected runtime behaviors. Compiled PCODE contains references to symbol tables or even memory addresses (i.e. macrocompiled codeblocks). If application stores such codeblock in some external holder, i.e. memo file then after restoring by different instance of the same application memory addresses can be different so for macrocompiled codeblocks it's not safe at all. For codeblocks compiled by compiler it can work until exactly the same application is used. But any modifications in the code can change symbol table so after restoring the result can be unpredictable. A codeblock like:

```
{|x| qout( x ) }
```

can be defined in code where just after QOUT() in symbol table is FERASE(). Small modification in application code can cause that symbols will be moved and above codeblock after restoring will effectively execute FERASE() instead of QOUT(). What does it mean for programs which makes something like:

```
eval( cb, "The most import file is:" )
eval( cb, cDataFile )
eval( cb, "Please make backup." )
```

I do not have to explain.

With full respect to xHarbour users IMO such "extension" is giving knife into monkey hands.

Codeblock serialization can be implemented in safe way but it needs at least some partial PCODE (de|re)compiler. If Harbour users find codeblock serialization as important extension then maybe we create such PCODE recompiler and such functionality will be added. Anyhow it will be necessary to keep recompiler code synced with compiler and it will be additional job in any future compiler modifications which can change generated PCODE.

## NATIVE RDDs

Both compilers shares most of RDD code. The differences are only in some recently added features to both compilers like timestamp values or MT mode support which are in xHarbour not fully implemented yet. Some of the most recent modifications in xHarbour core code are local to xHarbour only and I hope will never be part of Harbour core code. In xHarbour RDDs starting with "RE" prefix, i.e. REDBFCDX are simple copy of original RDDs where hb_file*() functions are replaced with hb_fileNet*() so these are the same RDDs but with different file transfer layer. Instead of standard OS file IO they use dedicate file server on TCP IP socket.

In Harbour it's possible to dynamically register many alternative RDD IO APIs and use them simultaneously in one application. HBNETIO library in Harbour implements this so without touching even single line in RDD code it works for all core RDDs and also 3-rd party ones if they use Harbour RDD IO API (hb_file*() functions). Additionally client and server code in HBNETIO are fully MT safe so can be used in MT programs without any problems.

Harbour has also similar library called HBMEMIO which allows to use native RDDs with pseudo files stored in memory instead of real files.

This feature was recently copied from Harbour to xHarbour but I haven't tested if it works or not. For sure RE*DBF RDDs still exists.

In both compilers maximal file size for tables, memos and indexes is limited only by OS and file format structures. Neither Harbour nor xHarbour introduce own limits here.

The maximal file size for DBFs is limited by number of records 2^32-1 = 4294967295 and maximal record size: 2^16-1 = 65535 what gives nearly 2^48 = 256TB as maximal .dbf file size. The maximal memo format size depends on used memo type: DBT, FPT or SMT and size of memo block. It's limited by maximal number of memo blocks = 2^32 and size of memo block so it's 2^32*<size_of_memo_block>. The default memo block size for DBT is 512 bytes, FPT - 64 bytes and for SMT 32 bytes. So for standard memo block sizes the maximum are: DBT->2TB, FPT->256GB, SMT->128GB. The maximal memo block size in Harbour is 2^32 and minimal is 1 byte and it can be any value between 1 and 65536 and then any number of 64KB blocks. The last limitation is introduced as workaround for some wrongly implemented in other languages memo drivers which were setting only 16 bits in 32bit field in memo header. Most of other languages has limit for memo block size at 2^15 and the block size has to be power of 2. Some of them also introduce minimal block size limits. If programmers plans to share data with programs compiled by such languages then he should check their documentation to not create memo files which cannot be accessed by them.

Maximal NTX file size for standard NTX files is 4GB and it's limited by internal NTX structures. Enabling 64bit locking in [x]Harbour change slightly used NTX format and increase maximum NTX file size to 4TB. The NTX format in [x]Harbour has also many other extensions like support for multitag indexes or using record number as hidden part of index key and many others which are unique to [x]Harbour. In practice all of CDX extensions are supported by NTX in [x]Harbour. The NSX format in [x]Harbour is also

limited be default to 4GB but like in NTX enabling 64bit locking extend it to 4TB. It also supports common to NTX and CDX set of features.

The CDX format is limited to 4GB and so far [x]Harbour does not support extended mode which can increase the size up to 2TB with standard page length and it can be bigger in all formats if we introduce support for bigger index pages. Of course all such extended formats are not binary compatible with original ones and so far can be used only by [x]Harbour RDDs though in ADS the .adi format is such extended CDX format so maybe in the future it will be possible to use .adi indexes in our CDX RDD.

Of course all of the above sizes can be reduced by operating system (OS) or file system (FS) limitations so it's necessary to check what is supported by environment where [x]Harbour applications are executed.

## REGULAR EXPRESSIONS

Harbour and xHarbour support regular expressions. In xHarbour they are bound with HVM and always present. In Harbour they are fully optional. Harbour supports few different regular expression libraries and it's possible to use platform or CRTL native libs or PCRE which is included in Harbour. Harbour always uses native for given library API. xHarbour supports only PCRE modified to use POSIX regex interface which does not support strings with embedded chr(0) characters.

Both compilers can store and reuse compiled regular expressions. xHarbour stores them in string items and Harbour in GC pointer items. Portable code should respect these differences.

## INET SOCKETS

Both compilers support IP4 sockets with similar .prg API created by Giancarlo Niccolai though both use different low level implementations. In xHarbour IP4 functions have Inet*() prefix.

In Harbour they have hb_inet*() prefix in core code and xHarbour Inet*() functions are available in XHB library.

Harbour has also public C socket API (hbsocket.h) which is not reduced to IP4 protocols only and is the base low level code used for .prg level [hb_]inet*() function implementation. This socket implementation was rewritten from scratch, it's MT safe and was designed to hide platform differences in BSD socket implementation. It also works in DOS builds using WATT-32 library.

In the future Harbour will have new .prg level API for sockets based on Harbour C socket API.

# I18N SUPPORT

Harbour and xHarbour compilers have build-in support for internationalization (I18N) and it's enabled during compilation using the same compiler time switch -j[<file>] but the low level implementation in compilers and at runtime is completely different.

In xHarbour during compilation with -j switch compiler looks for all call like I18N( <cConstValue> ) collects them and then stores them using internal xHarbour binary format in file with .hil. The base name of such file is taken from compiler .prg module or set by user by optional <file> argument of -j switch. If file already exists then strings extracted from compiled code are always appended to existing file without merging or checking for duplicated strings. Duplicated strings are eliminated by 'hbdict' program which should be used to create translation table. This programs reads .hil file show all existing strings and allow to type translation for each existing string and then saves the translation table in files with .hit extension using another internal xHarbour binary format. At runtime application may load .hit file and then I18N() function will look if string passed as 1-st parameter exists in translation table and if yes then it returns translated string and if not it returns original string. Only pure 1 to 1 translation exists without any extensions like context domains, support for plural forms or automatic CP translations. Detail information about I18N in xHarbour can be found in xHarbour CVS in doc/hbi18n.txt file.

Harbour compiler is very close to 'gettext' functionality with some additional extensions. It recognize the following functions at compile time:

```
hb_i18n_gettext( <cText> [, <cDomain> ] )
hb_i18n_gettext_strict( <cText> [, <cDomain> ] )
hb_i18n_gettext_noop( <cText> [, <cDomain> ] )
hb_i18n_ngettext( <nCount>, <cText> [, <cDomain> ] )
hb_i18n_ngettext_strict( <nCount>, <cText> [, <cDomain> ] )
hb_i18n_ngettext_noop( <nCount>, <cText> [, <cDomain> ] )
```

and then generates .pot text files which are gettext compatible so it's possible to use gettext tools to process them (merge, translate, update, etc.). Additionally Harbour has own tool called hbi18n which can merge .pot files, add automatic translations from other .po[t] or .hbl files and generate compiled Harbour I18N binary modules as .hbl files. Harbour supports plural forms translations, context domains, automatic CP translations, etc. The plural form support in Harbour is extended in comparison to gettext and allows to use non US languages as base. In MT mode each thread inherits I18N translation module from parent thread but then can set and use its own one.

There is a set of HB_I18N_*() functions available for programmers at runtime which allows to make different operations on compiled I18N modules and also .po[t] files.

Using -w3 switch during compilation enable additional validation of hb_i18n_[n]gettext*() parameters so compiler generates warnings. Just like in original gettext it's suggested to use #define or #xtranslate macros instead of direct calls to hb_i18n_[n]gettext*() functions, i.e.:

```
#xtranslate _I( <x,...> ) => hb_i18n_gettext( <x> )
#xtranslate _IN( <x,...> ) => hb_i18n_ngettext( <x> )
```

or:

```
#xtranslate I18N( <x,...> ) => hb_i18n_gettext( <x> )
```

It allows to keep source code shorter and if necessary easy switch to STRICT (for strict parameter validation) or NOOP (disabled at compile time I18N support) versions.

Additionally Harbour compiler can recognize user I18N functions. They have the same name as above hb_i18n_*() functions but with additional user '_*' suffix so they are in the form like:

```
hb_i18n_[n]gettext{_strict,_noop,}_*([<params,...>])
```

Using them users can easy introduce their own I18N runtime modules. To reduce dependencies on external tools by default Harbour uses own format for compiled .po[t] files but it's planned to add also native runtime gettext support as optional user I18N interface.

## ZLIB

Harbour RTL gives support at .prg level to ZLIB (HB_Z*()) and GZIP (HB_GZ*()) functions. In xHarbour RTL only ZLIB compression/decompression is available with by: HB_COMPRESS(), HB_UNCOMPRESS(), HB_COMPRESSBUFLEN(), HB_COMPRESSERROR() and HB_COMPRESSERRORDESC().

In Harbour these functions are available in XHB library. Original Harbour ZLIB API is different. It does not have any non MT safe extensions, it's protected against possible overflows and it's more closer to original ZLIB one.

## MACRO COMPILER

In Harbour macro compiler supports the same expressions as compiler. The only one and documented difference is support for VFP like datetime constant values {^...} which are supported only by compiler. It's guaranteed that any valid expressions not longer then 8MB will be cleanly compiled and executed by Harbour. It's possible to compile and execute longer expressions (in practice there is no maximal size limit) but in such case it's not guaranteed that all expressions can be compiled, for example macro expressions containing string constant values longer then 16MB cannot be compiled. It's only guaranteed that if Harbour cannot compile macro expression then it generates RT error and never generates broken code.

In xHarbour there are differences between macro compiler and compiler and not all expressions supported at compile time can be used in macro compiler. The maximum size for valid expressions which are always correctly compiled by xHarbour is 32KB. Expressions longer then 32KB can be compiled without any RT error by xHarbour but it's possible that wrong code will be generated for them which may cause any unpredictable behavior so user should not use such expressions. There is no clean way in xHarbour to check if macro expression longer then 32KB will be correctly compiled.

Just like Clipper neither Harbour nor xHarbour support statements in macro compiler so extended codeblocks also cannot be compiled. Now such functionality is supported only by compiler library in Harbour.

In Clipper the documented maximum size of expression which can be compiled by macro compiler is 256. Sometime Clipper's macro compiler can compile a little bit longer expressions generating correct PCODE anyhow just like in xHarbour it may also cause some unpredictable results (i.e. application crash) though in most of cases it's RT error during macro compilation.

# COMPILER LIBRARY

Harbour has compiler library which allows to integrate Harbour compiler with user applications and use it at runtime to compile new code which can be also immediately executed or dynamically registered as part of code. In Harbour compiler code is fully MT safe so the compiler library can be safely used in MT programs, for example for parallel compilation. Now compiler library is on pure GPL license so any programs which wants to use it have to respect the GPL license too.

# PP LIBRARY

Harbour and xHarbour has PP library which allows to use pre processor in user .prg code. Current PP in both compilers are fully MT safe without any internal locks so it can be used concurrently in MT programs.

## LEXER

xHarbour uses SIMPLEX as lexer for compiler and macro compiler. Introducing this lexer was next reason of the xHarbour fork. Internally SIMPLEX uses a lot of static variables and C compile time macros to speed-up some operations. It's faster than lexer used before based on code generated by FLEX but it's not MT safe and very hard to debug due to code hidden by nested C macros. The reduced version of SIMPLEX is used by xHarbour in macro compiler.

The preprocessor and lexer are not integrated in xHarbour. The preprocessed code is converted from tokens used by PP to a string and then this string is divided to tokens again by SIMPLEX.

Harbour compiler accepts tokens used by preprocessor so in fact it does not have separate compiler lexer at all. The compiler lexer code (complex.c) is only simple translator of token values between PP and bison terminal symbols. For macro compiler Harbour by default uses small lexer (macrolex.c) which is optimized for speed but it can also use PP lexer when Harbour is compiled with HB_MACRO_PPLEX. All Harbour lexers are fully MT safe without any internal locks. They are also much faster then xHarbour ones.

# CONTRIB LIBRARIES

Some of xHarbour core libraries like CT, TIP or ODBC are supported by Harbour as contrib library. In practice Harbour covers whole xHarbour functionality with some fixes and extensions. In contrib tree Harbour has also many other libraries which are unique to Harbour, like HBCURL, RDDSQL, HBSSL, GTWVG, HBQT, HBXBP, ... which give many important extensions. Some of them can be easily ported to xHarbour but some others not due to not fully functional MT model in xHarbour or missing some important functionality like thread safe support for many GTs in single application and multi window GTs with runtime window switching. It's possible that in the future some important fixes and extensions will be done in xHarbour core code what allow to port some of Harbour contrib libraries to xHarbour.

Detail description of contrib libraries needs separate text and will be much longer then this whole document and maybe will be created in the future.

## PORTABILITY

Both compilers were ported to many different platforms on different hardware. The list of supported platforms is really long: different *nixes (Linux, MacOSX, SunOS, HP-UX, *BSD,...) on little and big endian 32 and 64 bit machines, DOS, OS2, Win32, Win64 and WinCE (Harbour only). In practice they can be ported quite easy to nearly each OS and hardware if it passes two important conditions: it's ASCII based machine and has support for 8bit bytes. Due to cleaner code it's easier to port Harbour to new platform then xHarbour but the difference seems to not be huge for someone who has enough knowledge about destination platform. Now all popular operating systems are supported by both compilers with the exception to WinCE which is not supported by xHarbour yet.

## C LEVEL COMPATIBILITY

The main difference between Harbour and xHarbour in public C API is in value quantifiers. Harbour fully respect 'const' variable attribute and does not remove it from returned values. It helps to keep code clean and locate places where string constant values are overwritten what is illegal. It also helps C compiler to better optimize the code because it has additional information that some variables/memory regions are readonly and cannot be modified during code execution what can give some addition speed improvement. If some C code does not respect it then during compilation with Harbour header files C compilers usually generate warning and C++ ones errors, i.e. code like:

```
char * pszValue = hb_parc( 1 );
```

is wrong because hb_parc() returns pointer to readonly strings which cannot be changed by user code. So this code should be changed to:

```
const char * pszValue = hb_parc( 1 );
```

Now if compiled code tries to make something like:

```
pszValue[ index ] = 'X';
```

what is illegal in Harbour, xHarbour and also in Clipper for pointers returned by _parc() function then compiler generate warning or error about writing to readonly location.

In summary it means that Harbour forces to keep code clean and use more strict declarations but xHarbour and Clipper don't and wrong code can be silently compiled without any warnings or errors.

The second important difference is in hb_par*() and hb_stor*() functions. In Harbour there are hb_parv*() and hb_storv*() functions which accepts parameters like in Clipper _par*() and _stor*() functions with optional array indexes, i.e.:

```
const char * pszValue = hb_parvc( 1, 1 );
int          iValue   = hb_parvni( 1, 2 );
```

and functions without 'v' which do not allow to access array items, i.e.:

```
const char * pszValue = hb_parc( 1 );
int          iValue   = hb_parni( 2 );
```

These functions (without 'v') are also much more restrictive on accepted parameters and do not make hidden parameter translations, i.e. hb_parl() returns TRUE only for logical parameters and does not accept numeric values like hb_parvl() or _parl().

New functions without array index support were introduced to safely access parameters without strict type checking. Such code:

```
HB_FUNC( NOT )
{
   int iValue = _parni( 1 );
   hb_retni( ~iValue );
```

```
}
```

is not fully safe when array parameter is passed from .prg level, i.e.

```
? NOT( { 5, 4, 7 } )
```

In such case the behavior is unpredictable because _parni() function tries to access from C function frame unexciting parameter with array index. Depending on used hardware (CPU) different things may happen. On some platforms _parni( 1 ) always return 0, on some other random value from the array and on some others with hardware function frame protection it generates exception and application crash. To make the above code safe programmer should change it to:

```
HB_FUNC( NOT )
{
   int iValue = ISNUM( 1 ) ? _parni( 1 ) : 0;
   hb_retni( ~iValue );
}
```

or:

```
HB_FUNC( NOT )
{
   int iValue = _parni( 1, 0 );
   hb_retni( ~iValue );
}
```

xHarbour does not have hb_par*() and hb_stor*() functions without optional array index parameters so they cannot be safely used to access parameters without type checking. It may also create problems when xHarbour code is moved to Harbour because hb_par*() and hb_stor*() functions in xHarbour work like hb_parv*() and hb_storv*() in Harbour. It can be seen in code like:

```
int iValue = hb_parni( 1, 1 );
```

C compiler refuses to compile such code with Harbour header files and it has to be modified to:

```
int iValue = hb_parvni( 1, 1 );
```

For portable code which access array items using hb_par/hb_stor API I suggest you use hb_parv*() and hb_storv*() functions and add in some header file:

```
#ifdef __XHARBOUR__
   #define hb_parvc        hb_parc
   #define hb_parvni       hb_parni
   [...]
   #define hb_storvc       hb_storc
   #define hb_storvni      hb_storni
   [...]
#endif
```

or use Clipper compatible functions defined in extend.api: _par*() and _stor*().

I hope that in the future xHarbour will have above #define translation in core code or

maybe even support for hb_parv*() and hb_storv*() functions.

Next important difference between Harbour and xHarbour is access to internal HVM structures. In Harbour internal HVM structures are hidden by default so programmers cannot access them, i.e. PHB_ITEM is mapped as 'void *' pointer so code like:

```
iValue = pItem->item.asInteger.value;
```

cannot be compiled and programmers should change it to use official API:

```
iValue = hb_itemGetNI( pItem );
```

It's very important for core developers because public and internal C API is separated so we can easy introduce modifications in HVM internals without worrying about backward compatibility and it allows to keep the same code for different HVM versions (i.e. for MT and ST modes) so in Harbour only HBVM library is different for MT mode and all others are the same.

Such separation also greatly increase C code quality eliminating possible GPF traps, wrong behavior in code created without enough knowledge about HVM internals or code which blocks adding new extensions.

It's something that should be done in xHarbour core code where lot of code is simply wrong due to direct access to HVM internals, i.e.

*source/rtl/txtline.c[299]:*

```
Term[i] = (char *) (&Opt)->item.asString.value;
```

and we have GPF trap if given array item is not a string.

*source/rtl/version[108]:*

```
PHB_ITEM pQuery = hb_param( 1, HB_IT_INTEGER );
[...]
if( pQuery )
{
   int iQuery = pQuery->item.asInteger.value;
   […]
```

so it does not work if passed numeric parameter is not internally stored as HB_IT_INTEGER value and programmer cannot control it yourself, i.e. parameters can be read from table or passed from remote client and then after deserialization stored as HB_IT_LONG or HB_IT_DOUBLE.

*source/rtl/direct.c[124]:*

```
HB_ITEM_NEW( SubDir );
hb_fsDirectory( &SubDir, szSubdir, szAttributes, FALSE, TRUE );
hb_fsDirectoryCrawler( &SubDir, pResult, szFName, szAttributes, sRegEx );
```

This code creates complex item (array) on C stack so garbage collector does
not know anything about it so it cannot be activated otherwise it will cause GPF. It means that this code blocks adding automatic garbage collector activation in memory manager.

These are only examples and anyone can find many other similar problems in xHarbour

core code so in my opinion API separation with core code cleanup is sth what has to be done in xHarbour in the future.

Of course such separation does not mean that we can forbid 3-rd party programmers to access HVM internals. Programmers creating code for Harbour can include before any other Harbour header files "hbvmint.h", i.e.:

```
#include "hbvmint.h"
#include "hbapi.h"
```

and it enable access to HVM internals anyhow in such case we do not guaranty that such code will work in the future Harbour versions so we strongly recommend to not use it.

In Harbour core code only HBVM library is compiled with access to HVM internals. All other code uses official API. In contrib code "hbvmint.h" is used in few places in XHB library to emulate some core HVM xHarbour extensions.

In xHarbour there are few functions which allows to create string items without terminating '\0' characters like: hb_itemPutCRaw(), hb_itemPutCRawStatic(), hb_retclenAdoptRaw(). Because string items without terminating '\0' character exploit problems in any C code which needs ASCIIZ string creating possible GPF traps in all str*() and similar functions then in Harbour such functionality is illegal and will never be added to core code. Additionally in Harbour functions corresponding to xHarbour functions: hb_retclenStatic(). hb_itemPutCLStatic() have additional protection against creating such strings and generate internal error when programmer tries to pass wrong strings without trailing '\0' character. It may exploit problems in wrong code ported to Harbour.

Most of other public C functions in Harbour and xHarbour are compatible anyhow there are some differences i.e. in hash array or timestamp API which have to be covered by #ifdef __XHARBOUR__ conditional compilation. Programmers creating portable code have to know about them. For people who want to use only extended API for code used for Clipper and [x]Harbour I suggest to use small definition:

```
#ifndef __HARBOUR__
   #define HB_FUNC( fun )  CLIPPER fun ( void )
#endif
```

This definition allows to easy create code which can be compiled with Clipper and Harbour or xHarbour header files (extend.api).

It can be even added to Clipper's extend.api and then we can compile this code:

```
#include "extend.api"
HB_FUNC( NOT )
{
   int iValue = _parni( 1, 0 );
   hb_retni( ~iValue );
}
```

for Clipper, Harbour and xHarbour. Please only remember that linkers used with [x]Harbour are case sensitive and always use upper letters for function names. In Clipper it was not necessary.

## HBRUN / XBSCRIPT

xHarbour has .prg code interpreter mostly written also as .prg code. It's called xbscript. AFAIK It's also based for commercial product xBaseScript distributed by xHarbour.com.

In Harbour the same functionality was reached in just few lines in a program called hbrun because it's possible to use compiler library. HBRUN can compile any given code just like Harbour Compiler (in fact it uses the same compiler code from compiler library) and then dynamically execute it.

Unlike XBSCRIPT HBRUN does cause any performance reduction in comparison to standalone compilation with Harbour compiler to PCODE (-gc[0-2], -gh) because exactly the same PCODE is generated and later executed in both cases. Also the compilation phase is many times faster then in XBSCRIPT. The commercial version of xBaseScript has some extensions which allows to integrate it with IE and/or IIS. HBRUN from Harbour does not have such functionality yet.

## HBMK2

It's very nice new tool in Harbour which hides differences between platforms and C compiler used with Harbour. It should help new users to start working with Harbour (i.e. we can create copies of hbmk2 executable file called 'clipper' and 'rtlink' or 'blinker' and then it's possible to use them with original Clipper make file projects (i.e. in .rmk files created for Clipper and RMAKE). hbmk2 also nicely helps in creating binaries for different platforms (cross compilation), i.e. in creating WinCE programs in Linux or desktop Windows.

There are many features supported by this tool which can greatly help new and advanced Harbour users. Their detail description is not this text goal. If someone is interested in list of supported options then he can use hbmk2 with '--help' parameter. In the nearest future hbmk2 should replace hb* scripts.

This tool is unique to Harbour and does not exist in xHarbour.

## PERFORMANCE AND RESOURCE USAGE

Harbour internal structures are optimized for memory usage and are smaller than xHarbour ones so total memory usage by Harbour is also smaller. The difference depends on type of code but usually Harbour needs about 20%-25% less memory then xHarbour. The size of final executable is also usually smaller due to reduced dependencies which are necessary for pure HVM code though in current days it's less important issue. Much bigger difference is in performance.

Both compiler shares a lot of common C code in some subsystems like RDDs so in many operations which needs a lot of CPU to execute the same C code the performance is nearly the same. The difference begins to be noticeable when we compare HVM performance and time necessary to execute .prg code. Harbour is ~75% faster in ST mode and over 100% faster in MT mode. In harbour/tests/speedtst.prg we have simple test which can be used to compare performance of different compilers. The tests below were done using AMD Phenom(tm) 8450 Triple-Core Processor 2100 MHZ with 32bit Linux kernels.

Here are results for ST mode:

```
2009.07.28 20:48:12 Linux 2.6.25.20-0.4-pae i686
Harbour 2.0.0beta2 (Rev. 11910) GNU C 4.4 (32-bit) x86
THREADS: 0
N_LOOPS: 1000000
[ T000: empty loop overhead ].................................0.03
======================================================================
[ T001: x := L_C ]............................................0.02
[ T002: x := L_N ]............................................0.02
[ T003: x := L_D ]............................................0.02
[ T004: x := S_C ]............................................0.05
[ T005: x := S_N ]............................................0.03
[ T006: x := S_D ]............................................0.03
[ T007: x := M->M_C ].........................................0.06
[ T008: x := M->M_N ].........................................0.04
[ T009: x := M->M_D ].........................................0.04
[ T010: x := M->P_C ].........................................0.06
[ T011: x := M->P_N ].........................................0.04
[ T012: x := M->P_D ].........................................0.04
[ T013: x := F_C ]............................................0.15
[ T014: x := F_N ]............................................0.25
[ T015: x := F_D ]............................................0.10
[ T016: x := o:Args ].........................................0.14
[ T017: x := o[2] ]...........................................0.10
[ T018: round( i / 1000, 2 ) ]................................0.21
[ T019: str( i / 1000 ) ].....................................0.55
[ T020: val( s ) ]............................................0.28
[ T021: val( a [ i % 16 + 1 ] ) ].............................0.39
[ T022: dtos( d - i % 10000 ) ]...............................0.33
[ T023: eval( { || i % 16 } ) ]...............................0.36
[ T024: eval( bc := { || i % 16 } ) ].........................0.20
[ T025: eval( { |x| x % 16 }, i ) ]...........................0.29
[ T026: eval( bc := { |x| x % 16 }, i ) ].....................0.20
[ T027: eval( { |x| f1( x ) }, i ) ]..........................0.31
[ T028: eval( bc := { |x| f1( x ) }, i ) ].....................0.24
[ T029: eval( bc := &("{ |x| f1( x ) }"), i ) ]...............0.23
[ T030: x := &( "f1(" + str(i) + ")" ) ].....................2.95
```

```
[ T031: bc := &( "{|x|f1(x)}" ), eval( bc, i ) ].............2.97
[ T032: x := valtype( x ) +  valtype( i ) ].................0.34
[ T033: x := strzero( i % 100, 2 ) $ a[ i % 16 + 1 ] ].......0.62
[ T034: x := a[ i % 16 + 1 ] == s ].........................0.21
[ T035: x := a[ i % 16 + 1 ] = s ].........................0.23
[ T036: x := a[ i % 16 + 1 ] >= s ].........................0.23
[ T037: x := a[ i % 16 + 1 ] <= s ].........................0.23
[ T038: x := a[ i % 16 + 1 ] < s ].........................0.23
[ T039: x := a[ i % 16 + 1 ] > s ].........................0.23
[ T040: ascan( a, i % 16 ) ]................................0.24
[ T041: ascan( a, { |x| x == i % 16 } ) ]...................2.34
[ T042: iif( i%1000==0, a:={}, ) , aadd(a,{i,1,.T.,s,s2,a2 ]....0.70
[ T043: x := a ]............................................0.04
[ T044: x := {} ]...........................................0.12
[ T045: f0() ]..............................................0.06
[ T046: f1( i ) ]...........................................0.10
[ T047: f2( c[1...8] ) ]....................................0.10
[ T048: f2( c[1...40000] ) ]................................0.09
[ T049: f2( @c[1...40000] ) ]...............................0.09
[ T050: f2( @c[1...40000] ), c2 := c ]......................0.11
[ T051: f3( a, a2, s, i, s2, bc, i, n, x ) ]................0.31
[ T052: f2( a ) ]...........................................0.11
[ T053: x := f4() ].........................................0.45
[ T054: x := f5() ].........................................0.22
[ T055: x := space(16) ]....................................0.17
[ T056: f_prv( c ) ]........................................0.31
======================================================================
[ total application time: ].................................20.27
[ total real time: ]........................................20.51

2009.07.28 20:48:46 Linux 2.6.25.20-0.4-pae i686
xHarbour build 1.2.1 Intl. (SimpLex) (Rev. 6517) GNU C 4.4 (32 bit) ?
THREADS: 0
N_LOOPS: 1000000
[ T000: empty loop overhead ]..............................0.09
======================================================================
[ T001: x := L_C ]..........................................0.02
[ T002: x := L_N ]..........................................0.00
[ T003: x := L_D ]..........................................0.02
[ T004: x := S_C ]..........................................0.00
[ T005: x := S_N ]..........................................0.01
[ T006: x := S_D ]..........................................0.01
[ T007: x := M->M_C ].......................................0.02
[ T008: x := M->M_N ].......................................0.02
[ T009: x := M->M_D ].......................................0.01
[ T010: x := M->P_C ].......................................0.02
[ T011: x := M->P_N ].......................................0.06
[ T012: x := M->P_D ].......................................0.01
[ T013: x := F_C ]..........................................0.18
[ T014: x := F_N ]..........................................0.20
[ T015: x := F_D ]..........................................0.10
[ T016: x := o:Args ].......................................0.27
[ T017: x := o[2] ].........................................0.05
[ T018: round( i / 1000, 2 ) ]..............................0.30
[ T019: str( i / 1000 ) ]...................................0.69
[ T020: val( s ) ]..........................................0.34
[ T021: val( a [ i % 16 + 1 ] ) ]...........................0.51
[ T022: dtos( d - i % 10000 ) ].............................0.44
[ T023: eval( { || i % 16 } ) ].............................0.54
[ T024: eval( bc := { || i % 16 } ) ]........................0.36
```

```
[ T025: eval( { |x| x % 16 }, i ) ].............................0.43
[ T026: eval( bc := { |x| x % 16 }, i ) ].......................0.33
[ T027: eval( { |x| f1( x ) }, i ) ]............................0.58
[ T028: eval( bc := { |x| f1( x ) }, i ) ].....................0.48
[ T029: eval( bc := &("{ |x| f1( x ) }"), i ) ].................0.51
[ T030: x := &( "f1(" + str(i) + ")" ) ].......................4.70
[ T031: bc := &( "{|x|f1(x)}" ), eval( bc, i ) ]...............5.62
[ T032: x := valtype( x ) +  valtype( i ) ].....................0.64
[ T033: x := strzero( i % 100, 2 ) $ a[ i % 16 + 1 ] ].........0.87
[ T034: x := a[ i % 16 + 1 ] == s ].............................0.28
[ T035: x := a[ i % 16 + 1 ] = s ].............................0.28
[ T036: x := a[ i % 16 + 1 ] >= s ].............................0.27
[ T037: x := a[ i % 16 + 1 ] <= s ].............................0.28
[ T038: x := a[ i % 16 + 1 ] < s ].............................0.27
[ T039: x := a[ i % 16 + 1 ] > s ].............................0.28
[ T040: ascan( a, i % 16 ) ]....................................0.47
[ T041: ascan( a, { |x| x == i % 16 } ) ].......................3.95
[ T042: iif( i%1000==0, a:={}, ) , aadd(a,{i,1,.T.,s,s2,a2 ]....0.89
[ T043: x := a ]................................................0.01
[ T044: x := {} ]...............................................0.11
[ T045: f0() ]..................................................0.14
[ T046: f1( i ) ]...............................................0.19
[ T047: f2( c[1...8] ) ]........................................0.19
[ T048: f2( c[1...40000] ) ]....................................0.18
[ T049: f2( @c[1...40000] ) ]...................................0.18
[ T050: f2( @c[1...40000] ), c2 := c ]..........................0.22
[ T051: f3( a, a2, s, i, s2, bc, i, n, x ) ]....................0.39
[ T052: f2( a ) ]...............................................0.18
[ T053: x := f4() ].............................................0.81
[ T054: x := f5() ].............................................0.49
[ T055: x := space(16) ]........................................0.33
[ T056: f_prv( c ) ]............................................0.75
====================================================================
[ total application time: ].....................................34.52
[ total real time: ]............................................34.96
```

And here are results for MT mode:

```
2009.07.28 20:52:43 Linux 2.6.25.20-0.4-pae i686
Harbour 2.0.0beta2 (Rev. 11910) (MT) GNU C 4.4 (32-bit) x86
THREADS: 0
N_LOOPS: 1000000
[ T000: empty loop overhead ]..................................0.04
====================================================================
[ T001: x := L_C ]..............................................0.07
[ T002: x := L_N ]..............................................0.02
[ T003: x := L_D ]..............................................0.02
[ T004: x := S_C ]..............................................0.05
[ T005: x := S_N ]..............................................0.03
[ T006: x := S_D ]..............................................0.04
[ T007: x := M->M_C ]...........................................0.05
[ T008: x := M->M_N ]...........................................0.05
[ T009: x := M->M_D ]...........................................0.05
[ T010: x := M->P_C ]...........................................0.05
[ T011: x := M->P_N ]...........................................0.05
[ T012: x := M->P_D ]...........................................0.04
[ T013: x := F_C ]..............................................0.16
[ T014: x := F_N ]..............................................0.24
[ T015: x := F_D ]..............................................0.10
```

```
[ T016: x := o:Args ].....................................0.15
[ T017: x := o[2] ].......................................0.09
[ T018: round( i / 1000, 2 ) ]............................0.22
[ T019: str( i / 1000 ) ].................................0.62
[ T020: val( s ) ]........................................0.28
[ T021: val( a [ i % 16 + 1 ] ) ].........................0.41
[ T022: dtos( d - i % 10000 ) ]...........................0.39
[ T023: eval( { || i % 16 } ) ]...........................0.44
[ T024: eval( bc := { || i % 16 } ) ].....................0.27
[ T025: eval( { |x| x % 16 }, i ) ].......................0.38
[ T026: eval( bc := { |x| x % 16 }, i ) ].................0.24
[ T027: eval( { |x| f1( x ) }, i ) ].....................0.40
[ T028: eval( bc := { |x| f1( x ) }, i ) ].................0.29
[ T029: eval( bc := &("{ |x| f1( x ) }"), i ) ]...........0.31
[ T030: x := &( "f1(" + str(i) + ")" ) ]..................2.83
[ T031: bc := &( "{|x|f1(x)}" ), eval( bc, i ) ]..........3.16
[ T032: x := valtype( x ) +  valtype( i ) ]...............0.38
[ T033: x := strzero( i % 100, 2 ) $ a[ i % 16 + 1 ] ]....0.75
[ T034: x := a[ i % 16 + 1 ] == s ].......................0.28
[ T035: x := a[ i % 16 + 1 ] = s ]........................0.29
[ T036: x := a[ i % 16 + 1 ] >= s ].......................0.29
[ T037: x := a[ i % 16 + 1 ] <= s ].......................0.29
[ T038: x := a[ i % 16 + 1 ] < s ]........................0.29
[ T039: x := a[ i % 16 + 1 ] > s ]........................0.29
[ T040: ascan( a, i % 16 ) ]..............................0.30
[ T041: ascan( a, { |x| x == i % 16 } ) ].................2.87
[ T042: iif( i%1000==0, a:={}, ) , aadd(a,{i,1,.T.,s,s2,a2 ]....0.81
[ T043: x := a ]..........................................0.03
[ T044: x := {} ].........................................0.13
[ T045: f0() ]............................................0.07
[ T046: f1( i ) ].........................................0.11
[ T047: f2( c[1...8] ) ]..................................0.11
[ T048: f2( c[1...40000] ) ]..............................0.11
[ T049: f2( @c[1...40000] ) ].............................0.11
[ T050: f2( @c[1...40000] ), c2 := c ]....................0.14
[ T051: f3( a, a2, s, i, s2, bc, i, n, x ) ]..............0.33
[ T052: f2( a ) ].........................................0.11
[ T053: x := f4() ].......................................0.45
[ T054: x := f5() ].......................................0.24
[ T055: x := space(16) ]..................................0.19
[ T056: f_prv( c ) ]......................................0.38
======================================================================
[ total application time: ]..............................23.10
[ total real time: ].....................................23.78

2009.07.28 20:53:24 Linux 2.6.25.20-0.4-pae i686
xHarbour build 1.2.1 Intl. (SimpLex) (Rev. 6517) (MT) GNU C 4.4 (32 bit) ?
THREADS: 0
N_LOOPS: 1000000
[ T000: empty loop overhead ]............................0.20
======================================================================
[ T001: x := L_C ]........................................0.00
[ T002: x := L_N ]........................................0.00
[ T003: x := L_D ]........................................0.00
[ T004: x := S_C ]........................................0.00
[ T005: x := S_N ]........................................0.00
[ T006: x := S_D ]........................................0.00
[ T007: x := M->M_C ].....................................0.19
[ T008: x := M->M_N ].....................................0.23
[ T009: x := M->M_D ].....................................0.18
```

```
[ T010: x := M->P_C ].....................................0.24
[ T011: x := M->P_N ].....................................0.24
[ T012: x := M->P_D ].....................................0.21
[ T013: x := F_C ]........................................0.17
[ T014: x := F_N ]........................................0.21
[ T015: x := F_D ]........................................0.07
[ T016: x := o:Args ].....................................0.36
[ T017: x := o[2] ].......................................0.05
[ T018: round( i / 1000, 2 ) ]............................0.43
[ T019: str( i / 1000 ) ].................................0.83
[ T020: val( s ) ]........................................0.41
[ T021: val( a [ i % 16 + 1 ] ) ].........................0.65
[ T022: dtos( d - i % 10000 ) ]...........................0.56
[ T023: eval( { || i % 16 } ) ]...........................0.78
[ T024: eval( bc := { || i % 16 } ) ].....................0.47
[ T025: eval( { |x| x % 16 }, i ) ].......................0.66
[ T026: eval( bc := { |x| x % 16 }, i ) ].................0.49
[ T027: eval( { |x| f1( x ) }, i ) ].....................0.88
[ T028: eval( bc := { |x| f1( x ) }, i ) ].................0.73
[ T029: eval( bc := &("{ |x| f1( x ) }"), i ) ]...........0.72
[ T030: x := &( "f1(" + str(i) + ")" ) ]..................5.69
[ T031: bc := &( "{|x|f1(x)}" ), eval( bc, i ) ]..........6.18
[ T032: x := valtype( x ) +  valtype( i ) ]...............0.79
[ T033: x := strzero( i % 100, 2 ) $ a[ i % 16 + 1 ] ]....1.12
[ T034: x := a[ i % 16 + 1 ] == s ].......................0.37
[ T035: x := a[ i % 16 + 1 ] = s ]........................0.36
[ T036: x := a[ i % 16 + 1 ] >= s ].......................0.42
[ T037: x := a[ i % 16 + 1 ] <= s ].......................0.36
[ T038: x := a[ i % 16 + 1 ] < s ]........................0.37
[ T039: x := a[ i % 16 + 1 ] > s ]........................0.38
[ T040: ascan( a, i % 16 ) ]..............................0.64
[ T041: ascan( a, { |x| x == i % 16 } ) ].................6.21
[ T042: iif( i%1000==0, a:={}, ) , aadd(a,{i,1,.T.,s,s2,a2 ]....1.29
[ T043: x := a ]..........................................0.00
[ T044: x := {} ].........................................0.15
[ T045: f0() ]............................................0.21
[ T046: f1( i ) ].........................................0.26
[ T047: f2( c[1...8] ) ]..................................0.27
[ T048: f2( c[1...40000] ) ]..............................0.27
[ T049: f2( @c[1...40000] ) ].............................0.25
[ T050: f2( @c[1...40000] ), c2 := c ]....................0.33
[ T051: f3( a, a2, s, i, s2, bc, i, n, x ) ]..............0.57
[ T052: f2( a ) ].........................................0.27
[ T053: x := f4() ].......................................0.99
[ T054: x := f5() ].......................................0.65
[ T055: x := space(16) ]..................................0.38
[ T056: f_prv( c ) ]......................................1.24
=============================================================
[ total application time: ]..............................50.90
[ total real time: ].....................................51.49
```

Please note that in xHarbour MT mode many important things resolved in Harbour have not been implemented so far so it's hard to say how may look final performance.

Recently I've repeated above tests on the same Hardware but using 64-bit Linux kernels and 64bit Harbour builds and here the difference is even bigger. Here are results for ST mode:

```
2009.10.27 13:24:41 Linux 2.6.31.3-1-desktop x86_64
```

```
Harbour 2.0.0beta3 (Rev. 12776) GNU C 4.4 (64-bit) x86-64
THREADS: 0
N_LOOPS: 1000000
[ T000: empty loop overhead ].................................0.02
======================================================================
[ T001: x := L_C ]............................................0.03
[ T002: x := L_N ]............................................0.02
[ T003: x := L_D ]............................................0.02
[ T004: x := S_C ]............................................0.03
[ T005: x := S_N ]............................................0.03
[ T006: x := S_D ]............................................0.03
[ T007: x := M->M_C ].........................................0.03
[ T008: x := M->M_N ].........................................0.04
[ T009: x := M->M_D ].........................................0.03
[ T010: x := M->P_C ].........................................0.04
[ T011: x := M->P_N ].........................................0.03
[ T012: x := M->P_D ].........................................0.03
[ T013: x := F_C ]............................................0.10
[ T014: x := F_N ]............................................0.17
[ T015: x := F_D ]............................................0.08
[ T016: x := o:Args ].........................................0.11
[ T017: x := o[2] ]...........................................0.06
[ T018: round( i / 1000, 2 ) ]................................0.15
[ T019: str( i / 1000 ) ].....................................0.32
[ T020: val( s ) ]............................................0.18
[ T021: val( a [ i % 16 + 1 ] ) ].............................0.28
[ T022: dtos( d - i % 10000 ) ]...............................0.28
[ T023: eval( { || i % 16 } ) ]...............................0.29
[ T024: eval( bc := { || i % 16 } ) ].........................0.18
[ T025: eval( { |x| x % 16 }, i ) ]...........................0.24
[ T026: eval( bc := { |x| x % 16 }, i ) ].....................0.18
[ T027: eval( { |x| f1( x ) }, i ) ]..........................0.27
[ T028: eval( bc := { |x| f1( x ) }, i ) ].....................0.22
[ T029: eval( bc := &("{ |x| f1( x ) }"), i ) ]...............0.22
[ T030: x := &( "f1(" + str(i) + ")" ) ].....................2.05
[ T031: bc := &( "{|x|f1(x)}" ), eval( bc, i ) ].............2.41
[ T032: x := valtype( x ) +  valtype( i ) ]...................0.27
[ T033: x := strzero( i % 100, 2 ) $ a[ i % 16 + 1 ] ].......0.54
[ T034: x := a[ i % 16 + 1 ] == s ]...........................0.17
[ T035: x := a[ i % 16 + 1 ] = s ]............................0.19
[ T036: x := a[ i % 16 + 1 ] >= s ]...........................0.18
[ T037: x := a[ i % 16 + 1 ] <= s ]...........................0.19
[ T038: x := a[ i % 16 + 1 ] < s ]............................0.19
[ T039: x := a[ i % 16 + 1 ] > s ]............................0.18
[ T040: ascan( a, i % 16 ) ]..................................0.22
[ T041: ascan( a, { |x| x == i % 16 } ) ].....................2.01
[ T042: iif( i%1000==0, a:={}, ) , aadd(a,{i,1,.T.,s,s2,a2 ]....0.53
[ T043: x := a ]..............................................0.02
[ T044: x := {} ].............................................0.08
[ T045: f0() ]................................................0.05
[ T046: f1( i ) ].............................................0.08
[ T047: f2( c[1...8] ) ]......................................0.07
[ T048: f2( c[1...40000] ) ]..................................0.07
[ T049: f2( @c[1...40000] ) ].................................0.08
[ T050: f2( @c[1...40000] ), c2 := c ].......................0.09
[ T051: f3( a, a2, s, i, s2, bc, i, n, x ) ].................0.21
[ T052: f2( a ) ].............................................0.08
[ T053: x := f4() ]...........................................0.34
[ T054: x := f5() ]...........................................0.16
[ T055: x := space(16) ].....................................0.12
```

```
[ T056: f_prv( c ) ].....................................0.21
========================================================================
[ total application time: ].....................................15.60
[ total real time: ].....................................15.61

2009.10.27 13:25:10 Linux 2.6.31.3-1-desktop x86_64
xHarbour build 1.2.1 Intl. (SimpLex) (Rev. 6629) GNU C 4.4 (64 bit) ?
THREADS: 0
N_LOOPS: 1000000
[ T000: empty loop overhead ].....................................0.06
========================================================================
[ T001: x := L_C ].....................................0.05
[ T002: x := L_N ].....................................0.03
[ T003: x := L_D ].....................................0.04
[ T004: x := S_C ].....................................0.05
[ T005: x := S_N ].....................................0.03
[ T006: x := S_D ].....................................0.05
[ T007: x := M->M_C ].....................................0.05
[ T008: x := M->M_N ].....................................0.04
[ T009: x := M->M_D ].....................................0.04
[ T010: x := M->P_C ].....................................0.06
[ T011: x := M->P_N ].....................................0.04
[ T012: x := M->P_D ].....................................0.04
[ T013: x := F_C ].....................................0.25
[ T014: x := F_N ].....................................0.17
[ T015: x := F_D ].....................................0.11
[ T016: x := o:Args ].....................................0.30
[ T017: x := o[2] ].....................................0.07
[ T018: round( i / 1000, 2 ) ].....................................0.32
[ T019: str( i / 1000 ) ].....................................0.87
[ T020: val( s ) ].....................................0.32
[ T021: val( a [ i % 16 + 1 ] ) ].....................................0.48
[ T022: dtos( d - i % 10000 ) ].....................................0.43
[ T023: eval( { || i % 16 } ) ].....................................0.62
[ T024: eval( bc := { || i % 16 } ) ].....................................0.40
[ T025: eval( { |x| x % 16 }, i ) ].....................................0.47
[ T026: eval( bc := { |x| x % 16 }, i ) ].....................................0.37
[ T027: eval( { |x| f1( x ) }, i ) ].....................................0.66
[ T028: eval( bc := { |x| f1( x ) }, i ) ].....................................0.56
[ T029: eval( bc := &("{ |x| f1( x ) }"), i ) ].....................................0.58
[ T030: x := &( "f1(" + str(i) + ")" ) ].....................................5.95
[ T031: bc := &( "{|x|f1(x)}" ), eval( bc, i ) ].....................................6.57
[ T032: x := valtype( x ) +  valtype( i ) ].....................................0.76
[ T033: x := strzero( i % 100, 2 ) $ a[ i % 16 + 1 ] ].....................................0.90
[ T034: x := a[ i % 16 + 1 ] == s ].....................................0.25
[ T035: x := a[ i % 16 + 1 ] = s ].....................................0.25
[ T036: x := a[ i % 16 + 1 ] >= s ].....................................0.24
[ T037: x := a[ i % 16 + 1 ] <= s ].....................................0.25
[ T038: x := a[ i % 16 + 1 ] < s ].....................................0.25
[ T039: x := a[ i % 16 + 1 ] > s ].....................................0.24
[ T040: ascan( a, i % 16 ) ].....................................0.46
[ T041: ascan( a, { |x| x == i % 16 } ) ].....................................4.10
[ T042: iif( i%1000==0, a:={}, ) , aadd(a,{i,1,.T.,s,s2,a2 ]....0.92
[ T043: x := a ].....................................0.04
[ T044: x := {} ].....................................0.16
[ T045: f0() ].....................................0.21
[ T046: f1( i ) ].....................................0.23
[ T047: f2( c[1...8] ) ].....................................0.25
[ T048: f2( c[1...40000] ) ].....................................0.24
[ T049: f2( @c[1...40000] ) ].....................................0.24
```

```
[ T050: f2( @c[1...40000] ), c2 := c ].........................0.28
[ T051: f3( a, a2, s, i, s2, bc, i, n, x ) ]...................0.41
[ T052: f2( a ) ]..............................................0.24
[ T053: x := f4() ]............................................0.94
[ T054: x := f5() ]............................................0.63
[ T055: x := space(16) ].......................................0.41
[ T056: f_prv( c ) ]...........................................0.86
================================================================
[ total application time: ]...................................37.14
[ total real time: ]..........................................37.16
```

Harbour is noticeable faster in 64 bit mode and xHarbour slower but I haven't analyzed the reasons. Few years ago xHarbour was also faster in 64bit modes so it's probably problem with some compile time settings.

Harbour build system allows to use different compile time switches for static and shared libraries and the bigger difference is probably result of better tuned switches for static libraries which cannot be used for shared ones so xHarbour cannot use them by default.

The last test we can make is scalability in real multi CPU environment.

```
2009.07.28 21:07:56 Linux 2.6.25.20-0.4-pae i686
Harbour 2.0.0beta2 (Rev. 11910) (MT)+ GNU C 4.4 (32-bit) x86
THREADS: 2
N_LOOPS: 1000000
                                                          1 th.  2 th.  factor
================================================================================
[ T001: x := L_C ]_____________________________________  0.19   0.07 ->  2.65
[ T002: x := L_N ]_____________________________________  0.12   0.06 ->  1.97
[ T003: x := L_D ]_____________________________________  0.12   0.06 ->  1.97
[ T004: x := S_C ]_____________________________________  0.21   0.26 ->  0.81
[ T005: x := S_N ]_____________________________________  0.15   0.08 ->  1.99
[ T006: x := S_D ]_____________________________________  0.16   0.08 ->  2.00
[ T007: x := M->M_C ]__________________________________  0.19   0.13 ->  1.42
[ T008: x := M->M_N ]__________________________________  0.18   0.09 ->  1.98
[ T009: x := M->M_D ]__________________________________  0.21   0.14 ->  1.51
[ T010: x := M->P_C ]__________________________________  0.18   0.13 ->  1.45
[ T011: x := M->P_N ]__________________________________  0.17   0.09 ->  1.89
[ T012: x := M->P_D ]__________________________________  0.17   0.09 ->  2.00
[ T013: x := F_C ]_____________________________________  0.44   0.30 ->  1.48
[ T014: x := F_N ]_____________________________________  0.57   0.32 ->  1.81
[ T015: x := F_D ]_____________________________________  0.32   0.17 ->  1.94
[ T016: x := o:Args ]__________________________________  0.44   0.24 ->  1.81
[ T017: x := o[2] ]____________________________________  0.29   0.16 ->  1.83
[ T018: round( i / 1000, 2 ) ]_________________________  0.59   0.28 ->  2.12
[ T019: str( i / 1000 ) ]______________________________  1.21   0.69 ->  1.74
[ T020: val( s ) ]_____________________________________  0.61   0.33 ->  1.86
[ T021: val( a [ i % 16 + 1 ] ) ]______________________  0.93   0.50 ->  1.85
[ T022: dtos( d - i % 10000 ) ]________________________  0.90   0.56 ->  1.60
[ T023: eval( { || i % 16 } ) ]________________________  1.00   1.24 ->  0.81
[ T024: eval( bc := { || i % 16 } ) ]__________________  0.59   0.32 ->  1.83
[ T025: eval( { |x| x % 16 }, i ) ]____________________  0.81   1.03 ->  0.79
[ T026: eval( bc := { |x| x % 16 }, i ) ]______________  0.59   0.30 ->  1.96
[ T027: eval( { |x| f1( x ) }, i ) ]___________________  0.93   1.10 ->  0.85
[ T028: eval( bc := { |x| f1( x ) }, i ) ]_____________  0.71   0.38 ->  1.89
[ T029: eval( bc := &("{ |x| f1( x ) }"), i ) ]________  0.70   0.40 ->  1.74
[ T030: x := &( "f1(" + str(i) + ")" ) ]_______________  5.71   3.36 ->  1.70
[ T031: bc := &( "{|x|f1(x)}" ), eval( bc, i ) ]_______  6.36   4.21 ->  1.51
```

```
[ T032: x := valtype( x ) +  valtype( i ) ]____________  0.86   0.51 ->  1.68
[ T033: x := strzero( i % 100, 2 ) $ a[ i % 16 + 1 ] ]  1.52   0.85 ->  1.80
[ T034: x := a[ i % 16 + 1 ] == s ]___________________  0.62   0.35 ->  1.80
[ T035: x := a[ i % 16 + 1 ] = s ]____________________  0.67   0.38 ->  1.75
[ T036: x := a[ i % 16 + 1 ] >= s ]___________________  0.66   0.37 ->  1.76
[ T037: x := a[ i % 16 + 1 ] <= s ]___________________  0.66   0.38 ->  1.73
[ T038: x := a[ i % 16 + 1 ] < s ]____________________  0.67   0.38 ->  1.75
[ T039: x := a[ i % 16 + 1 ] > s ]____________________  0.65   0.33 ->  1.98
[ T040: ascan( a, i % 16 ) ]__________________________  0.67   0.33 ->  2.01
[ T041: ascan( a, { |x| x == i % 16 } ) ]_____________  5.87   3.63 ->  1.62
[ T042: iif( i%1000==0, a:={}, ) , aadd(a,{i,1,.T.,s ]  1.88   1.45 ->  1.29
[ T043: x := a ]______________________________________  0.15   0.07 ->  1.96
[ T044: x := {} ]_____________________________________  0.37   1.00 ->  0.37
[ T045: f0() ]________________________________________  0.23   0.14 ->  1.58
[ T046: f1( i ) ]_____________________________________  0.31   0.20 ->  1.57
[ T047: f2( c[1...8] ) ]______________________________  0.32   0.18 ->  1.80
[ T048: f2( c[1...40000] ) ]__________________________  0.31   0.17 ->  1.78
[ T049: f2( @c[1...40000] ) ]_________________________  0.30   0.21 ->  1.40
[ T050: f2( @c[1...40000] ), c2 := c ]________________  0.36   0.24 ->  1.51
[ T051: f3( a, a2, s, i, s2, bc, i, n, x ) ]__________  0.74   0.42 ->  1.78
[ T052: f2( a ) ]_____________________________________  0.31   0.16 ->  1.95
[ T053: x := f4() ]___________________________________  0.96   0.57 ->  1.66
[ T054: x := f5() ]___________________________________  0.59   0.39 ->  1.53
[ T055: x := space(16) ]______________________________  0.48   0.32 ->  1.50
[ T056: f_prv( c ) ]__________________________________  0.86   0.56 ->  1.53
================================================================================
[   TOTAL   ]_____________________________________________ 46.76  30.75 ->  1.52
================================================================================
[ total application time: ].................................106.45
[ total real time: ]........................................77.52


2009.07.28 21:11:44 Linux 2.6.25.20-0.4-pae i686
xHarbour build 1.2.1 Intl. (SimpLex) (Rev. 6517) (MT)+ GNU C 4.4 (32 bit) ?
THREADS: 2
N_LOOPS: 1000000
                                                          1 th.  2 th.  factor
================================================================================
[ T001: x := L_C ]____________________________________  0.46   0.20 ->  2.33
[ T002: x := L_N ]____________________________________  0.37   0.18 ->  2.04
[ T003: x := L_D ]____________________________________  0.38   0.22 ->  1.68
[ T004: x := S_C ]____________________________________  0.42   0.38 ->  1.10
[ T005: x := S_N ]____________________________________  0.36   0.21 ->  1.69
[ T006: x := S_D ]____________________________________  0.39   0.19 ->  2.07
[ T007: x := M->M_C ]_________________________________  1.39   1.22 ->  1.14
[ T008: x := M->M_N ]_________________________________  1.42   1.12 ->  1.27
[ T009: x := M->M_D ]_________________________________  1.44   1.16 ->  1.24
[ T010: x := M->P_C ]_________________________________  1.46   1.27 ->  1.15
[ T011: x := M->P_N ]_________________________________  1.43   1.10 ->  1.30
[ T012: x := M->P_D ]_________________________________  1.40   1.10 ->  1.27
[ T013: x := F_C ]____________________________________  0.76   0.41 ->  1.84
[ T014: x := F_N ]____________________________________  0.76   0.43 ->  1.76
[ T015: x := F_D ]____________________________________  0.53   0.28 ->  1.87
[ T016: x := o:Args ]_________________________________  1.04   0.96 ->  1.09
[ T017: x := o[2] ]___________________________________  0.51   0.25 ->  2.04
[ T018: round( i / 1000, 2 ) ]________________________  1.15   0.98 ->  1.17
[ T019: str( i / 1000 ) ]_____________________________  1.97   1.51 ->  1.30
[ T020: val( s ) ]____________________________________  1.19   0.86 ->  1.38
[ T021: val( a [ i % 16 + 1 ] ) ]_____________________  1.65   1.23 ->  1.35
[ T022: dtos( d - i % 10000 ) ]_______________________  1.54   1.14 ->  1.35
```

```
[ T023: eval( { || i % 16 } ) ]_____________________________  1.91   2.78 ->  0.68
[ T024: eval( bc := { || i % 16 } ) ]_____________________  1.46   1.34 ->  1.09
[ T025: eval( { |x| x % 16 }, i ) ]_______________________  1.60   2.77 ->  0.58
[ T026: eval( bc := { |x| x % 16 }, i ) ]_________________  1.32   1.30 ->  1.02
[ T027: eval( { |x| f1( x ) }, i ) ]______________________  2.13   3.10 ->  0.69
[ T028: eval( bc := { |x| f1( x ) }, i ) ]________________  1.75   1.78 ->  0.98
[ T029: eval( bc := &("{ |x| f1( x ) }"), i ) ]___________  1.77   1.79 ->  0.99
[ T030: x := &( "f1(" + str(i) + ")" ) ]__________________ 11.34  14.11 ->  0.80
[ T031: bc := &( "{|x|f1(x)}" ), eval( bc, i ) ]__________ 13.19  17.26 ->  0.76
[ T032: x := valtype( x ) +  valtype( i ) ]_______________  2.11   1.65 ->  1.28
[ T033: x := strzero( i % 100, 2 ) $ a[ i % 16 + 1 ] ]____  2.80   1.78 ->  1.57
[ T034: x := a[ i % 16 + 1 ] == s ]_______________________  1.17   0.69 ->  1.71
[ T035: x := a[ i % 16 + 1 ] = s ]________________________  1.21   0.57 ->  2.10
[ T036: x := a[ i % 16 + 1 ] >= s ]_______________________  1.15   0.62 ->  1.87
[ T037: x := a[ i % 16 + 1 ] <= s ]_______________________  1.10   0.60 ->  1.86
[ T038: x := a[ i % 16 + 1 ] < s ]________________________  1.11   0.63 ->  1.75
[ T039: x := a[ i % 16 + 1 ] > s ]________________________  1.14   0.61 ->  1.88
[ T040: ascan( a, i % 16 ) ]______________________________  1.66   1.17 ->  1.41
[ T041: ascan( a, { |x| x == i % 16 } ) ]_________________ 12.39  13.63 ->  0.91
[ T042: iif( i%1000==0, a:={}, ) , aadd(a,{i,1,.T.,s ]____  2.95   2.79 ->  1.06
[ T043: x := a ]__________________________________________  0.37   0.19 ->  1.94
[ T044: x := {} ]_________________________________________  0.70   2.29 ->  0.31
[ T045: f0() ]____________________________________________  0.78   0.73 ->  1.07
[ T046: f1( i ) ]_________________________________________  0.91   0.87 ->  1.05
[ T047: f2( c[1...8] ) ]__________________________________  0.92   0.85 ->  1.08
[ T048: f2( c[1...40000] ) ]______________________________  0.91   0.84 ->  1.08
[ T049: f2( @c[1...40000] ) ]_____________________________  0.89   0.83 ->  1.07
[ T050: f2( @c[1...40000] ), c2 := c ]____________________  1.02   0.90 ->  1.13
[ T051: f3( a, a2, s, i, s2, bc, i, n, x ) ]______________  1.50   1.14 ->  1.32
[ T052: f2( a ) ]_________________________________________  0.92   0.85 ->  1.08
[ T053: x := f4() ]_______________________________________  2.45   1.95 ->  1.26
[ T054: x := f5() ]_______________________________________  1.75   1.75 ->  1.00
[ T055: x := space(16) ]__________________________________  1.19   1.02 ->  1.16
[ T056: f_prv( c ) ]______________________________________  4.51   4.36 ->  1.03
================================================================================
[   TOTAL   ]_____________________________________________106.08 105.94 ->  1.00
================================================================================
[ total application time: ].................................287.58
[ total real time: ]........................................212.02
```

As we can see in xHarbour we do not have any improvement executing MT programs on MultiCPU machines while in Harbour the speed is noticeably better. It means that Harbour is quite well scalable and users should expect speed improvement executing MT parallel programs on multi CPU machines. For some programs like MT servers it may be critical – programs compiled by Harbour can be quite well improved by simple hardware upgrade to 4, 8, 16 or more CPU machines.