

Package 'bit64'

February 20, 2012

Type Package

Title A S3 class for vectors of 64bit integers

Version 0.8-4

Date 2011-12-12

Author Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

Maintainer Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

Depends R (>= 2.14.0), bit (>= 1.1-8), methods

Description Package 'bit64' provides serializable S3 atomic 64bit (signed) integers that can be used in vectors, matrices, arrays and data.frames. Methods are available for coercion from and to logicals, integers, doubles, characters as well as many elementwise and summary functions.

License GPL-2

LazyLoad yes

ByteCompile yes

URL <http://ff.r-forge.r-project.org/>

Encoding latin1

Repository CRAN

Repository/R-Forge/Project ff

Repository/R-Forge/Revision 97

Date/Publication 2012-02-20 10:04:32

R topics documented:

bit64-package	2
as.character.integer64	14
as.data.frame.integer64	15
as.integer64.character	16
c.integer64	17
cumsum.integer64	18
extract.replace.integer64	19
format.integer64	20
identical.integer64	21
rep.integer64	22
seq.integer64	23
sum.integer64	24
xor.integer64	25
Index	27

bit64-package	<i>A S3 class for vectors of 64bit integers</i>
---------------	---

Description

Package 'bit64' provides fast serializable S3 atomic 64bit (signed) integers that can be used in vectors, matrices, arrays and data.frames. Methods are available for coercion from and to logicals, integers, doubles, characters as well as many elementwise and summary functions.

With 'integer64' vectors you can store very large integers at the expense of 64 bits, which is by factor 7 better than 'int64' from package 'int64'. Due to the smaller memory footprint, the atomic vector architecture and using only S3 instead of S4 classes, most operations are one to three orders of magnitude faster: Example speedups are 4x for serialization, 250x for adding, 900x for coercion and 2000x for object creation. Also 'integer64' avoids an ongoing (potentially infinite) penalty for garbage collection observed during existence of 'int64' objects (see code in example section).

Last but not least, this package has no commercial copyright attached, it is not sponsored by any commercial company or otherwise influenced by commercial interests. Its mix of high-level R code with low-level C-code protects against misuse outside the GPLed R context.

Usage

```
integer64(length)
## S3 method for class 'integer64'
is(x)
## S3 replacement method for class 'integer64'
length(x) <- value
## S3 method for class 'integer64'
print(x, quote=FALSE, ...)
```

Arguments

length	length of vector using <code>integer</code>
x	an integer64 vector
value	an integer64 vector of values to be assigned
quote	logical, indicating whether or not strings should be printed with surrounding quotes.
...	further arguments to the <code>NextMethod</code>

Details

```

Package: bit64
Type: Package
Version: 0.5.0
Date: 2011-12-12
License: GPL-2
LazyLoad: yes
Encoding: latin1

```

Value

`integer64` returns a vector of 'integer64', i.e. a vector of `double` decorated with class 'integer64'.

Design considerations

64 bit integers are related to big data: we need them to overcome address space limitations. Therefore performance of the 64 bit integer type is critical. In the S language – designed in 1975 – atomic objects were defined to be vectors for a couple of good reasons: simplicity, option for implicit parallelization, good cache locality. In recent years many analytical databases have learnt that lesson: column based data bases provide superior performance for many applications, the result are products such as MonetDB, Sybase IQ, Vertica, Exasol, Ingres Vectorwise. If we introduce 64 bit integers not natively in Base R but as an external package, we should at least strive to make them as 'basic' as possible. Therefore the design choice of `bit64` not only differs from `int64`, it is obvious: Like the other atomic types in Base R, we model data type 'integer64' as a contiguous `atomic` vector in memory, and we use the more basic `S3` class system, not `S4`. Like package `int64` we want our 'integer64' to be `serializeable`, therefore we also use an existing data type as the basis. Again the choice is obvious: R has only one 64 bit data type: doubles. By using `doubles`, `integer64` inherits some functionality such as `is.atomic`, `length`, `length<-`, `names`, `names<-`, `dim`, `dim<-`, `dimnames`, `dimnames`.

Our R level functions strictly follow the functional programming paradigm: no modification of arguments or other sideeffects. However, internally we deviate from the strict paradigm in order to boost performance. Our C functions do not create new return values, instead we pass-in the memory to be returned as an argument. This gives us the freedom to apply the C-function to new or old vectors, which helps to avoid unnecessary memory allocation, unnecessary copying and unnecessary

garbage collection. *Within* our R functions we also deviate from conventional R programming by not using `attr<-` and `attributes<-` because they always do new memory allocation and copying. If we want to set attributes of return values that we have freshly created, we instead use functions `setattr` and `setattributes` from package `bit`. If you want to see both tricks at work, see method `integer64`.

Arithmetic precision and coercion

The fact that we introduce 64 bit long long integers – without introducing 128-bit long doubles – creates some subtle challenges: Unlike 32 bit `integers`, the `integer64` are no longer a proper subset of `double`. If a binary arithmetic operation does involve a `double` and a `integer`, it is a no-brainer to return `double` without loss of information. If an `integer64` meets a `double`, it is not trivial what type to return. Switching to `integer64` limits our ability to represent very large numbers, switching to `integer64` limits our ability to distinguish x from $x+1$. Since the latter is purpose of introducing 64 bit integers, we usually return `integer64` from functions involving `integer64`, for example in `c`, `cbind` and `rbind`.

Different from Base R, our operators `+`, `-`, `%/%` and `%` coerce their arguments to `integer64` and always return `integer64`.

The multiplication operator `*` coerces its first argument to `integer64` but allows its second argument to be also `double`: the second argument is internally coerced to 'long double' and the result of the multiplication is returned as `integer64`.

The division `/` and power `^` operators also coerce their first argument to `integer64` and coerce internally their second argument to 'long double', they return as `double`, like `sqrt`, `log`, `log2` and `log10` do.

Creating and testing S3 class 'integer64'

Our creator function `integer64` takes an argument `length`, creates an atomic double vector of this length, attaches an S3 class attribute 'integer64' to it, and that's it. We simply rely on S3 method dispatch and interpret those 64bit elements as 'long long int'.

`is.double` currently returns `TRUE` for `integer64` and might return `FALSE` in a later release. Consider `is.double` to have undefined behaviour and do query `is.integer64` *before* querying `is.double`.

The methods `is.integer64` and `is.vector` both return `TRUE` for `integer64`. Note that we did not patch `storage.mode` and `typeof`, which both continue returning 'double' Like for 32 bit `integer`, `mode` returns 'numeric' and `as.double` tries coercing to `double`). It is likely that 'integer64' becomes a `vmode` in package `ff`.

Further methods for creating `integer64` are `range` which returns the range of the data type if called without arguments, `rep`, `seq`.

For all available methods on `integer64` vectors see the index below and the examples.

Index of implemented methods

creating,testing,printing	see also	description
<code>integer64</code>	<code>integer</code>	create zero atomic vector
<code>rep.integer64</code>	<code>rep</code>	
<code>seq.integer64</code>	<code>seq</code>	
<code>is.integer64</code>	<code>is</code>	

is.vector.integer64	is.integer	inherited from Base R
identical.integer64	is.vector	
length<-.integer64	identical	
	length<-	inherited from Base R
	length	inherited from Base R
	names<-	inherited from Base R
	names	inherited from Base R
	dim<-	inherited from Base R
	dim	inherited from Base R
	dimnames<-	inherited from Base R
	dimnames	inherited from Base R
	str	inherited from Base R, does not print values correctly
print.integer64	print	
coercing to integer64	see also	description
as.integer64		generic
as.integer64.character	character	
as.integer64.double	double	
as.integer64.integer	integer	
as.integer64.integer64	integer64	
as.integer64.logical	logical	
as.integer64.NULL	NULL	
coercing from integer64	see also	description
as.bitstring	as.bitstring	generic
as.bitstring.integer64		
as.character.integer64	as.character	
as.double.integer64	as.double	
as.integer.integer64	as.integer	
as.logical.integer64	as.logical	
as.vector.integer64	as.vector	
data structures	see also	description
c.integer64	c	vector concatenate
cbind.integer64	cbind	column bind
rbind.integer64	rbind	row bind
as.data.frame.integer64	as.data.frame	coerce atomic object to data.frame
	data.frame	inherited from Base R since we have coercion
subscripting	see also	description
[.integer64	[vector and array extract
[<-.integer64	[<-	vector and array assign
[[.integer64	[[scalar extract
[[<-.integer64	[[<-	scalar assign
binary operators	see also	description
+.integer64	+	returns integer64
-.integer64	-	returns integer64

<code>*.integer64</code>	<code>*</code>	returns integer64
<code>^.integer64</code>	<code>^</code>	returns double
<code>/.integer64</code>	<code>/</code>	returns double
<code>%/.integer64</code>	<code>%/</code>	returns integer64
<code>%%.integer64</code>	<code>%%</code>	returns integer64
comparison operators	see also	description
<code>==.integer64</code>	<code>==</code>	
<code>!=.integer64</code>	<code>!=</code>	
<code><.integer64</code>	<code><</code>	
<code><=.integer64</code>	<code><=</code>	
<code>>.integer64</code>	<code>></code>	
<code>>=.integer64</code>	<code>>=</code>	
logical operators	see also	description
<code>!.integer64</code>	<code>!</code>	
<code>&.integer64</code>	<code>&</code>	
<code> .integer64</code>	<code> </code>	
<code>xor.integer64</code>	<code>xor</code>	
math functions	see also	description
<code>is.na.integer64</code>	<code>is.na</code>	returns logical
<code>format.integer64</code>	<code>format</code>	returns character
<code>abs.integer64</code>	<code>abs</code>	returns integer64
<code>sign.integer64</code>	<code>sign</code>	returns integer64
<code>log.integer64</code>	<code>log</code>	returns double
<code>log10.integer64</code>	<code>log10</code>	returns double
<code>log2.integer64</code>	<code>log2</code>	returns double
<code>sqrt.integer64</code>	<code>sqrt</code>	returns double
<code>ceiling.integer64</code>	<code>ceiling</code>	dummy returning its argument
<code>floor.integer64</code>	<code>floor</code>	dummy returning its argument
<code>trunc.integer64</code>	<code>trunc</code>	dummy returning its argument
<code>round.integer64</code>	<code>round</code>	dummy returning its argument
<code>signif.integer64</code>	<code>signif</code>	dummy returning its argument
cumulative functions	see also	description
<code>cummin.integer64</code>	<code>cummin</code>	
<code>cummax.integer64</code>	<code>cummax</code>	
<code>cumsum.integer64</code>	<code>cumsum</code>	
<code>cumprod.integer64</code>	<code>cumprod</code>	
<code>diff.integer64</code>	<code>diff</code>	
summary functions	see also	description
<code>range.integer64</code>	<code>range</code>	returns valid range without arguments
<code>min.integer64</code>	<code>min</code>	
<code>max.integer64</code>	<code>max</code>	
<code>sum.integer64</code>	<code>sum</code>	
<code>prod.integer64</code>	<code>prod</code>	

all.integer64	all	
any.integer64	any	
helper functions	see also	description
minusclass	minusclass	removing class attribute
plusclass	plusclass	inserting class attribute
binattr	binattr	define binary op behaviour
tested I/O functions	see also	description
	read.table	inherited from Base R
	write.table	inherited from Base R
	serialize	inherited from Base R
	unserialize	inherited from Base R
	save	inherited from Base R
	load	inherited from Base R
	dput	inherited from Base R
	dget	inherited from Base R

Limitations inherited from implementing 64 bit integers via an external package

- **vector size** of atomic vectors is still limited to `.Machine$integer.max`. However, external memory extending packages such as `ff` or `bigmemory` can extend their address space now with `integer64`. Having 64 bit integers also help with those not so obvious address issues that arise once we exchange data with SQL databases and datawarehouses, which use big integers as surrogate keys, e.g. on indexed primary key columns. This puts R into a relatively strong position compared to certain commercial statistical softwares, which sell database connectivity but neither have the range of 64 bit integers, nor have integers at all, nor have a single numeric data type in their macro-glue-language.
- **literals** such as `123LL` would require changes to Base R, up to then we need to write (and call) `as.integer64(123L)` or `as.integer64(123)` or `as.integer64('123')`. Only the latter allows to specify numbers beyond Base R's numeric data types and therefore is the recommended way to use – using only one way may facilitate migrating code to literals at a later stage.

Limitations inherited from Base R, Core team, can you change this?

- `identical` with default parameters does not distinguish all bit-patterns of doubles. For testing purposes we provide a wrapper `identical.integer64` that will distinguish all bit-patterns. It would be desirable to have a single call of `identical` handle both, `double` and `integer64`.
- the **colon** operator `:` officially does not dispatches S3 methods, therefore we have not patched it. However, the following code seems to work:

```

"::.default" <- get("::")
"::" <- function(from,to)UseMethod("::")
"::.integer64" <- function(from, to)seq.integer64(from=from, to=to)

```

Try for example:

```
from <- lim.integer64()[1]
to <- from+99
from:to
```

- `is.double` does not dispatches S3 methods, therefore we have not patched it. However, the following code seems to work if you want `is.double` to return `FALSE` on `integer64`:

```
if (!exists("is.double.default")){
  is.double.default <- function(x) base::is.double(x)
  is.double <- function(x)UseMethod("is.double")
}
is.double.integer64 <- function(x)FALSE
```

- `c` only dispatches `c.integer64` if the first argument is `integer64` and it does not recursively dispatch the proper method when called with argument `recursive=TRUE` Therefore

```
c(list(integer64, integer64))
```

does not work and for now you can only call

```
c.integer64(list(x,x))
```

- **generic binary operators** fail to dispatch *any* user-defined S3 method if the two arguments have two different S3 classes. For example we have two classes `bit` and `bitwhich` sparsely representing boolean vectors and we have methods `&.bit` and `&.bitwhich`. For an expression involving both as in `bit & bitwhich`, none of the two methods is dispatched. Instead a standard method is dispatched, which neither handles `bit` nor `bitwhich`. Although it lacks symmetry, the better choice would be to dispatch simply the method of the class of the first argument in case of class conflict. This choice would allow authors of extension packages providing coherent behaviour at least within their contributed classes. But as long as none of the package authors methods is dispatched, he cannot handle the conflicting classes at all.
- `unlist` is not generic and if it were, we would face similar problems as with `c()`
- `vector` with argument `mode='integer64'` cannot work without adjustment of Base R
- `as.vector` with argument `mode='integer64'` cannot work without adjustment of Base R
- `is.vector` does not dispatch its method `is.vector.integer64`
- `mode<-` drops the class `'integer64'` which is returned from `as.integer64`. Also it does not remove an existing class `'integer64'` when assigning mode `'integer'`.
- `storage.mode<-` does not support external data types such as `as.integer64`
- `matrix` does drop the `'integer64'` class attribute, while `array` works correctly with `integer64`
- `str` does not print the values of `integer64` correctly

Limitations planned to be removed with the next release

- `sort` is not yet implemented
- `order` is not yet implemented
- `match` is not yet implemented
- `duplicated` is not yet implemented
- `unique` is not yet implemented
- `table` is not yet implemented
- `as.factor` is not yet implemented

further limitations

- **subscripting** non-existing elements and subscripting with NAs is currently not supported. Such subscripting currently returns 9218868437227407266 instead of NA (the NA value of the underlying double code). Following the full R behaviour here would either destroy performance or require extensive C-coding.

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com> Maintainer: Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.c

See Also

`integer` in base R and `int64` in package 'int64'

Examples

```
message("Using integer64 in vector")
x <- integer64(8) # create 64 bit vector
x
is.atomic(x)     # TRUE
is.integer64(x)  # TRUE
is.numeric(x)    # TRUE
is.integer(x)    # FALSE - debatable
is.double(x)     # FALSE - might change
x[] <- 1:2        # assigned value is recycled as usual
x[1:6]           # subscripting as usual
length(x) <- 13  # changing length as usual
x
rep(x, 2)        # replicate as usual
seq(as.integer64(1), 10) # seq.integer64 is dispatched on first given argument
seq(to=as.integer64(10), 1) # seq.integer64 is dispatched on first given argument
seq.integer64(along.with=x) # or call seq.integer64 directly
x <- c(x,runif(length(x), max=100)) # c.integer64 is dispatched only if *first* argument is integer64 ...
x # ... and coerces everything to integer64 - including double
names(x) <- letters # use names as usual
x

message("Using integer64 in array - note that 'matrix' currently does not work")
y <- array(as.integer64(NA), dim=c(3,4), dimnames=list(letters[1:3], LETTERS[1:4]))
```

```

y["a",] <- 1:2      # assigning as usual
y
y[1:2,-4]         # subscripting as usual
cbind(E=1:3, F=runif(3, 0, 100), G=c("-1","0","1"), y) # cbind.integer64 dispatched on any argument and coerces ev

message("Using integer64 in data.frame")
str(as.data.frame(x))
str(as.data.frame(y))
str(data.frame(y))
str(data.frame(I(y)))
d <- data.frame(x=x, y=runif(length(x), 0, 100))
d
d$x

message("Using integer64 with csv files")
fi64 <- tempfile()
write.csv(d, file=fi64, row.names=FALSE)
e <- read.csv(fi64, colClasses=c("integer64", NA))
unlink(fi64)
str(e)
identical.integer64(d$x,e$x)

message("Serializing and unserializing integer64")
dput(d, fi64)
e <- dget(fi64)
identical.integer64(d$x,e$x)
e <- d[,]
save(e, file=fi64)
rm(e)
load(file=fi64)
identical.integer64(d,e)

### A couple of unit tests follow hidden in a dontshow{} directive ###

## Not run:
message("== Differences between integer64 and int64 ==")
require(bit64)
require(int64)

message("-- integer64 is atomic --")
is.atomic(integer64())
is.atomic(int64())
str(integer64(3))
str(int64(3))

message("-- The following performance numbers are measured under RWin64 --")
message("-- under RWin32 the advantage of integer64 over int64 is smaller --")

message("-- integer64 needs 7x/5x less RAM than int64 under 64/32 bit OS (and twice the RAM of integer as it should b
as.vector(object.size(int64(1e6))/object.size(integer64(1e6)))
as.vector(object.size(integer64(1e6))/object.size(integer(1e6)))

```

```

message("-- integer64 creates 2000x/1300x faster than int64 under 64/32 bit OS (and 3x the time of integer) --")
t32 <- system.time(integer(1e8))
t64 <- system.time(integer64(1e8))
T64 <- system.time(int64(1e7))*10 # using 1e8 as above stalls our R on an i7 8 GB RAM Thinkpad
T64/t64
t64/t32

i32 <- sample(1e6)
d64 <- as.double(i32)

message("-- the following timings are rather conservative since timings of integer64 include garbage collection --")
message("-- integer64 coerces 900x/100x faster than int64 under 64/32 bit OS (and 2x the time of coercing to integer) --")
t32 <- system.time(for(i in 1:1000)as.integer(d64))
t64 <- system.time(for(i in 1:1000)as.integer64(d64))
T64 <- system.time(as.int64(d64))*1000
T64/t64
t64/t32
td64 <- system.time(for(i in 1:1000)as.double(i32))
t64 <- system.time(for(i in 1:1000)as.integer64(i32))
T64 <- system.time(for(i in 1:10)as.int64(i32))*100
T64/t64
t64/td64

message("-- integer64 serializes 4x/0.8x faster than int64 under 64/32 bit OS (and less than 2x/6x the time of integer) --")
t32 <- system.time(for(i in 1:10)serialize(i32, NULL))
td64 <- system.time(for(i in 1:10)serialize(d64, NULL))
i64 <- as.integer64(i32);
t64 <- system.time(for(i in 1:10)serialize(i64, NULL))
rm(i64); gc()
I64 <- as.int64(i32);
T64 <- system.time(for(i in 1:10)serialize(I64, NULL))
rm(I64); gc()
T64/t64
t64/t32
t64/td64

message("-- integer64 adds 250x/60x faster than int64 under 64/32 bit OS (and less than 6x the time of integer or double) --")
td64 <- system.time(for(i in 1:100)d64+d64)
t32 <- system.time(for(i in 1:100)i32+i32)
i64 <- as.integer64(i32);
t64 <- system.time(for(i in 1:100)i64+i64)
rm(i64); gc()
I64 <- as.int64(i32);
T64 <- system.time(for(i in 1:10)I64+I64)*10
rm(I64); gc()
T64/t64
t64/t32
t64/td64

message("-- integer64 sums 3x/0.2x faster than int64 (and at about 5x/60X the time of integer and double) --")
td64 <- system.time(for(i in 1:100)sum(d64))
t32 <- system.time(for(i in 1:100)sum(i32))

```

```

i64 <- as.integer64(i32);
t64 <- system.time(for(i in 1:100)sum(i64))
rm(i64); gc()
I64 <- as.int64(i32);
T64 <- system.time(for(i in 1:100)sum(I64))
rm(I64); gc()
T64/t64
t64/t32
t64/td64

message("-- integer64 diffs 5x/0.85x faster than integer and double (int64 version 1.0 does not support diff) --")
td64 <- system.time(for(i in 1:10)diff(d64, lag=2L, differences=2L))
t32 <- system.time(for(i in 1:10)diff(i32, lag=2L, differences=2L))
i64 <- as.integer64(i32);
t64 <- system.time(for(i in 1:10)diff(i64, lag=2L, differences=2L))
rm(i64); gc()
t64/t32
t64/td64

message("-- integer64 subscripts 1000x/340x faster than int64 (and at the same speed / 10x slower as integer) --")
ts32 <- system.time(for(i in 1:1000)sample(1e6, 1e3))
t32<- system.time(for(i in 1:1000)i32[sample(1e6, 1e3)])
i64 <- as.integer64(i32);
t64 <- system.time(for(i in 1:1000)i64[sample(1e6, 1e3)])
rm(i64); gc()
I64 <- as.int64(i32);
T64 <- system.time(for(i in 1:100)I64[sample(1e6, 1e3)])*10
rm(I64); gc()
(T64-ts32)/(t64-ts32)
(t64-ts32)/(t32-ts32)

message("-- integer64 assigns 200x/90x faster than int64 (and 50x/160x slower than integer) --")
ts32 <- system.time(for(i in 1:100)sample(1e6, 1e3))
t32 <- system.time(for(i in 1:100)i32[sample(1e6, 1e3)] <- 1:1e3)
i64 <- as.integer64(i32);
i64 <- system.time(for(i in 1:100)i64[sample(1e6, 1e3)] <- 1:1e3)
rm(i64); gc()
I64 <- as.int64(i32);
I64 <- system.time(for(i in 1:100)I64[sample(1e6, 1e3)] <- 1:1e3)*10
rm(I64); gc()
(T64-ts32)/(t64-ts32)
(t64-ts32)/(t32-ts32)

tdfi32 <- system.time(dfi32 <- data.frame(a=i32, b=i32, c=i32))
tdfsi32 <- system.time(dfi32[1e6:1,])
fi32 <- tempfile()
tdfwi32 <- system.time(write.csv(dfi32, file=fi32, row.names=FALSE))
tdfri32 <- system.time(read.csv(fi32, colClasses=rep("integer", 3)))
unlink(fi32)
rm(dfi32); gc()

```

```

i64 <- as.integer64(i32);
tdfi64 <- system.time(dfi64 <- data.frame(a=i64, b=i64, c=i64))
tdfsi64 <- system.time(dfi64[1e6:1,])
fi64 <- tempfile()
tdfwi64 <- system.time(write.csv(dfi64, file=fi64, row.names=FALSE))
tdfri64 <- system.time(read.csv(fi64, colClasses=rep("integer64", 3)))
unlink(fi64)
rm(i64, dfi64); gc()

I64 <- as.int64(i32);
tdfI64 <- system.time(dfI64<-data.frame(a=I64, b=I64, c=I64))
tdfsI64 <- system.time(dfI64[1e6:1,])
fI64 <- tempfile()
tdfwI64 <- system.time(write.csv(dfI64, file=fI64, row.names=FALSE))
tdfrI64 <- system.time(read.csv(fI64, colClasses=rep("int64", 3)))
unlink(fI64)
rm(I64, dfI64); gc()

message("-- integer64 coerces 40x/6x faster to data.frame than int64(and factor 1/9 slower than integer) --")
tdfI64/tdfi64
tdfi64/tdfi32
message("-- integer64 subscripts from data.frame 20x/2.5x faster than int64 (and 3x/13x slower than integer) --")
tdfsI64/tdfsi64
tdfsi64/tdfsi32
message("-- integer64 csv writes about 2x/0.5x faster than int64 (and about 1.5x/5x slower than integer) --")
tdfwI64/tdfwi64
tdfwi64/tdfwi32
message("-- integer64 csv reads about 3x/1.5 faster than int64 (and about 2x slower than integer) --")
tdfrI64/tdfri64
tdfri64/tdfri32

rm(i32, d64); gc()

message("-- investigating the impact on garbage collection: --")
message("-- the fragmented structure of int64 messes up R's RAM --")
message("-- and slows down R's gargbage collection just by existing --")

td32 <- double(21)
td32[1] <- system.time(d64 <- double(1e7))[3]
for (i in 2:11)td32[i] <- system.time(gc(), gcFirst=FALSE)[3]
rm(d64)
for (i in 12:21)td32[i] <- system.time(gc(), gcFirst=FALSE)[3]

t64 <- double(21)
t64[1] <- system.time(i64 <- integer64(1e7))[3]
for (i in 2:11)t64[i] <- system.time(gc(), gcFirst=FALSE)[3]
rm(i64)
for (i in 12:21)t64[i] <- system.time(gc(), gcFirst=FALSE)[3]

T64 <- double(21)
T64[1] <- system.time(I64 <- int64(1e7))[3]
for (i in 2:11)T64[i] <- system.time(gc(), gcFirst=FALSE)[3]

```

```

rm(I64)
for (i in 12:21)T64[i] <- system.time(gc(), gcFirst=FALSE)[3]

matplot(1:21, cbind(td32, t64, T64), pch=c("d","i","I"), log="y")

## End(Not run)

```

as.character.integer64

Coerce from integer64

Description

Methods to coerce integer64 to other atomic types. 'as.bitstring' coerces to a human-readable bit representation (strings of zeroes and ones). The methods [format](#), [as.character](#), [as.double](#), [as.logical](#), [as.integer](#) do what you would expect.

Usage

```

as.bitstring(x, ...)
## S3 method for class 'integer64'
as.bitstring(x, ...)
## S3 method for class 'integer64'
as.character(x, ...)
## S3 method for class 'integer64'
as.double(x, ...)
## S3 method for class 'integer64'
as.integer(x, ...)
## S3 method for class 'integer64'
as.logical(x, ...)

```

Arguments

x an integer64 vector
 ... further arguments to the [NextMethod](#)

Value

as.bitstring returns a string of .
 The other methods return atomic vectors of the expected types

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

[as.integer64.character integer64](#)

Examples

```
as.character(lim.integer64())
as.bitstring(lim.integer64())
```

```
as.data.frame.integer64
```

integer64: Coercing to data.frame column

Description

Coercing integer64 vector to data.frame.

Usage

```
## S3 method for class 'integer64'
as.data.frame(x, ...)
```

Arguments

x	an integer64 vector
...	passed to NextMethod as.data.frame after removing the 'integer64' class attribute

Details

'as.data.frame.integer64' is rather not intended to be called directly, but it is required to allow integer64 as data.frame columns.

Value

a one-column data.frame containing an integer64 vector

Note

This is currently very slow – any ideas for improvement?

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

[cbind.integer64](#) [as.vector.integer64](#) [integer64](#)

Examples

```
as.data.frame.integer64(as.integer64(1:12))
data.frame(a=1:12, b=as.integer64(1:12))
```

as.integer64.character

Coerce to integer64

Description

Methods to coerce from other atomic types to integer64.

Usage

```
as.integer64(x, ...)  
## S3 method for class 'integer64'  
as.integer64(x, ...)  
## S3 method for class 'NULL'  
as.integer64(x, ...)  
## S3 method for class 'character'  
as.integer64(x, ...)  
## S3 method for class 'double'  
as.integer64(x, ...)  
## S3 method for class 'integer'  
as.integer64(x, ...)  
## S3 method for class 'logical'  
as.integer64(x, ...)
```

Arguments

x an atomic vector
... further arguments to the [NextMethod](#)

Details

as.integer64.character is realized using C function strtoll which does not support scientific notation. Instead of '1e6' use '1000000'.

Value

The other methods return atomic vectors of the expected types

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

[as.character.integer64](#) [integer64](#)

Examples

```
as.integer64(as.character(lim.integer64()))
```

c.integer64	<i>Concatenating integer64 vectors</i>
-------------	--

Description

The usual functions 'c', 'cbind' and 'rbind'

Usage

```
## S3 method for class 'integer64'  
c(..., recursive = FALSE)  
## S3 method for class 'integer64'  
cbind(...)  
## S3 method for class 'integer64'  
rbind(...)
```

Arguments

...	two or more arguments coerced to 'integer64' and passed to NextMethod
recursive	logical. If recursive = TRUE, the function recursively descends through lists (and pairlists) combining all their elements into a vector.

Value

[c](#) returns a integer64 vector of the total length of the input
[cbind](#) and [rbind](#) return a integer64 matrix

Note

R currently only dispatches generic 'c' to method 'c.integer64' if the first argument is 'integer64'

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

[rep.integer64](#) [seq.integer64](#) [as.data.frame.integer64](#) [integer64](#)

Examples

```
c(as.integer64(1), 2:6)  
cbind(1:6, as.integer(1:6))  
rbind(1:6, as.integer(1:6))
```

cumsum.integer64 *Cumulative Sums, Products, Extremes and lagged differences*

Description

Cumulative Sums, Products, Extremes and lagged differences

Usage

```
## S3 method for class 'integer64'  
cummin(x)  
## S3 method for class 'integer64'  
cummax(x)  
## S3 method for class 'integer64'  
cumsum(x)  
## S3 method for class 'integer64'  
cumprod(x)  
## S3 method for class 'integer64'  
diff(x, lag = 1L, differences = 1L, ...)
```

Arguments

x	an atomic vector of class 'integer64'
lag	see diff
differences	see diff
...	ignored

Value

[cummin](#), [cummax](#), [cumsum](#) and [cumprod](#) return a integer64 vector of the same length as their input
[diff](#) returns a integer64 vector shorter by lag*differences elements

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

[sum.integer64](#) [integer64](#)

Examples

```
cumsum(rep(as.integer64(1), 12))  
diff(as.integer64(c(0,1:12)))  
cumsum(as.integer64(c(0, 1:12)))  
diff(cumsum(as.integer64(c(0,0,1:12))), differences=2)
```

`extract.replace.integer64`*Extract or Replace Parts of an integer64 vector*

Description

Methods to extract and replace parts of an integer64 vector.

Usage

```
## S3 method for class 'integer64'  
x[...]  
## S3 replacement method for class 'integer64'  
x[...] <- value  
## S3 method for class 'integer64'  
x[[...]]  
## S3 replacement method for class 'integer64'  
x[[...]] <- value
```

Arguments

<code>x</code>	an atomic vector
<code>value</code>	an atomic vector with values to be assigned
<code>...</code>	further arguments to the NextMethod

Value

A vector or scalar of class 'integer64'

Note

You should not subscript non-existing elements and not use NAs as subscripts. The current implementation returns 9218868437227407266 instead of NA.

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

[[integer64](#)]

Examples

```
as.integer64(1:12)[1:3]
x <- as.integer64(1:12)
dim(x) <- c(3,4)
x
x[]
x[,2:3]
```

format.integer64

Unary operators and functions for integer64 vectors

Description

Unary operators and functions for integer64 vectors.

Usage

```
## S3 method for class 'integer64'
format(x, justify="right", ...)
## S3 method for class 'integer64'
is.na(x)
## S3 method for class 'integer64'
x
## S3 method for class 'integer64'
sign(x)
## S3 method for class 'integer64'
abs(x)
## S3 method for class 'integer64'
sqrt(x)
## S3 method for class 'integer64'
log(x, base)
## S3 method for class 'integer64'
log2(x)
## S3 method for class 'integer64'
log10(x)
## S3 method for class 'integer64'
floor(x)
## S3 method for class 'integer64'
ceiling(x)
## S3 method for class 'integer64'
trunc(x, ...)
## S3 method for class 'integer64'
round(x, digits=0)
## S3 method for class 'integer64'
signif(x, digits=6)
```

Arguments

x	an atomic vector of class 'integer64'
base	an atomic scalar (we save 50% log-calls by not allowing a vector base)
digits	integer indicating the number of decimal places (round) or significant digits (signif) to be used. Negative values are allowed (see round)
justify	should it be right-justified (the default), left-justified, centred or left alone.
...	further arguments to the NextMethod

Value

[format](#) returns a character vector
[is.na](#) and [!](#) return a logical vector
[sqrt](#), [log](#), [log2](#) and [log10](#) return a double vector
[sign](#), [abs](#), [floor](#), [ceiling](#), [trunc](#) and [round](#) return a vector of class 'integer64'
[signif](#) is not implemented

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

[xor.integer64](#) [integer64](#)

Examples

```
sqrt(as.integer64(1:12))
```

`identical.integer64` *Identity function for class 'integer64'*

Description

This will discover any deviation between objects containing integer64 vectors.

Usage

```
identical.integer64(x, y, num.eq = FALSE, single.NA = FALSE, attrib.as.set = TRUE, ignore.bytecode = TR
```

Arguments

x	atomic vector of class 'integer64'
y	atomic vector of class 'integer64'
num.eq	see identical
single.NA	see identical
attrib.as.set	see identical
ignore.bytecode	see identical

Details

This is simply a wrapper to `identical` with default arguments `num.eq = FALSE`, `single.NA = FALSE`.

Value

A single logical value, TRUE or FALSE, never NA and never anything other than a single value.

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

`==.integer64` `identical` `integer64`

Examples

```
i64 <- as.double(NA); class(i64) <- "integer64"
identical(i64-1, i64+1)
identical.integer64(i64-1, i64+1)
```

rep.integer64

Replicate elements of integer64 vectors

Description

Replicate elements of integer64 vectors

Usage

```
## S3 method for class 'integer64'
rep(x, ...)
```

Arguments

`x` a vector of 'integer64' to be replicated
`...` further arguments passed to `NextMethod`

Value

`rep` returns a integer64 vector

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

[c.integer64](#) [rep.integer64](#) [as.data.frame.integer64](#) [integer64](#)

Examples

```
rep(as.integer64(1:2), 6)
rep(as.integer64(1:2), c(6,6))
rep(as.integer64(1:2), length.out=6)
```

seq.integer64

integer64: Sequence Generation

Description

Generating sequence of integer64 values

Usage

```
## S3 method for class 'integer64'
seq(from = NULL, to = NULL, by = NULL, length.out = NULL, along.with = NULL, ...)
```

Arguments

from	integer64 scalar (in order to dispatch the integer64 method of seq)
to	scalar
by	scalar
length.out	scalar
along.with	scalar
...	ignored

Details

seq.integer64 does coerce its arguments 'from', 'to' and 'by' to integer64. If not provided, the argument 'by' is automatically determined as +1 or -1, but the size of 'by' is not calculated as in [seq](#) (because this might result in a non-integer value).

Value

an integer64 vector with the generated sequence

Note

In base R : currently is not generic and does not dispatch, see section "Limitations inherited from Base R" in [integer64](#)

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

[c.integer64](#) [rep.integer64](#) [as.data.frame.integer64](#) [integer64](#)

Examples

```
# colon not activated: as.integer64(1):12
seq(as.integer64(1), 12, 2)
seq(as.integer64(1), by=2, length.out=6)
```

sum.integer64

Summary functions for integer64 vectors

Description

Summary functions for integer64 vectors. Function 'range' without arguments returns the smallest and largest value of the 'integer64' class.

Usage

```
## S3 method for class 'integer64'
all(..., na.rm = FALSE)
## S3 method for class 'integer64'
any(..., na.rm = FALSE)
## S3 method for class 'integer64'
min(..., na.rm = FALSE)
## S3 method for class 'integer64'
max(..., na.rm = FALSE)
## S3 method for class 'integer64'
range(..., na.rm = FALSE)
lim.integer64()
## S3 method for class 'integer64'
sum(..., na.rm = FALSE)
## S3 method for class 'integer64'
prod(..., na.rm = FALSE)
```

Arguments

... atomic vectors of class 'integer64'
na.rm logical scalar indicating whether to ignore NAs

Details

The numerical summary methods always return `integer64`. Therefore the methods for `min`, `max` and `range` do not return `+Inf`, `-Inf` on empty arguments, but `+9223372036854775807`, `-9223372036854775807` (in this sequence). The same is true if only NAs are submitted with argument `na.rm=TRUE`.

`lim.integer64` returns these limits in proper order `-9223372036854775807`, `+9223372036854775807` and without a `warning`.

Value

`all` and `any` return a logical scalar

`range` returns a `integer64` vector with two elements

`min`, `max`, `sum` and `prod` return a `integer64` scalar

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

`cumsum.integer64` `integer64`

Examples

```
lim.integer64()  
range(as.integer64(1:12))
```

xor.integer64

Binary operators for integer64 vectors

Description

Binary operators for `integer64` vectors.

Usage

```
## S3 method for class 'integer64'  
e1 & e2  
## S3 method for class 'integer64'  
e1 | e2  
## S3 method for class 'integer64'  
xor(x,y)  
## S3 method for class 'integer64'  
e1 != e2  
## S3 method for class 'integer64'  
e1 == e2  
## S3 method for class 'integer64'  
e1 < e2
```

```
## S3 method for class 'integer64'  
e1 <= e2  
## S3 method for class 'integer64'  
e1 > e2  
## S3 method for class 'integer64'  
e1 >= e2  
## S3 method for class 'integer64'  
e1 + e2  
## S3 method for class 'integer64'  
e1 - e2  
## S3 method for class 'integer64'  
e1 * e2  
## S3 method for class 'integer64'  
e1 ^ e2  
## S3 method for class 'integer64'  
e1 / e2  
## S3 method for class 'integer64'  
e1 %% e2  
## S3 method for class 'integer64'  
e1 %% e2  
binattr(e1,e2) # for internal use only
```

Arguments

e1	an atomic vector of class 'integer64'
e2	an atomic vector of class 'integer64'
x	an atomic vector of class 'integer64'
y	an atomic vector of class 'integer64'

Value

&, |, xor, !=, ==, <, <=, >, >= return a logical vector
^ and / return a double vector
+, -, *, %/%, %% return a vector of class 'integer64'

Author(s)

Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>

See Also

[format.integer64](#) [integer64](#)

Examples

```
as.integer64(1:12) - 1
```

Index

!.integer64 (format.integer64), 20
!=.integer64 (xor.integer64), 25
*Topic **classes**
 as.character.integer64, 14
 as.data.frame.integer64, 15
 as.integer64.character, 16
 bit64-package, 2
 c.integer64, 17
 cumsum.integer64, 18
 extract.replace.integer64, 19
 format.integer64, 20
 identical.integer64, 21
 rep.integer64, 22
 seq.integer64, 23
 sum.integer64, 24
 xor.integer64, 25
*Topic **manip**
 as.character.integer64, 14
 as.data.frame.integer64, 15
 as.integer64.character, 16
 bit64-package, 2
 c.integer64, 17
 cumsum.integer64, 18
 extract.replace.integer64, 19
 format.integer64, 20
 identical.integer64, 21
 rep.integer64, 22
 seq.integer64, 23
 sum.integer64, 24
 xor.integer64, 25
*Topic **package**
 bit64-package, 2
*, 4, 6, 26
*.integer64, 6
*.integer64 (xor.integer64), 25
+, 4, 5, 26
+.integer64, 5
+.integer64 (xor.integer64), 25
-, 5
-.integer64, 5
-.integer64 (xor.integer64), 25
-, 4, 26
.Machine, 7
/, 4, 6, 26
/.integer64, 6
/.integer64 (xor.integer64), 25
:, 7, 23
<, 6, 26
<.integer64, 6
<.integer64 (xor.integer64), 25
<=, 6, 26
<=.integer64, 6
<=.integer64 (xor.integer64), 25
==, 6, 26
==.integer64, 6, 22
==.integer64 (xor.integer64), 25
>, 6, 26
>.integer64, 6
>.integer64 (xor.integer64), 25
>=, 6, 26
>=.integer64, 6
>=.integer64 (xor.integer64), 25
[, 5, 19
[.integer64, 5
[.integer64
 (extract.replace.integer64), 19
[<-, 5
[<-.integer64, 5
[<-.integer64
 (extract.replace.integer64), 19
[[, 5
[[.integer64, 5
[[.integer64
 (extract.replace.integer64), 19
[[<-, 5
[[<-.integer64, 5
[[<-.integer64
 (extract.replace.integer64), 19

- %%.integer64 (xor.integer64), 25
- %%.integer64 (xor.integer64), 25
- &, 6, 26
- &.bit, 8
- &.bitwhich, 8
- &.integer64, 6
- &.integer64 (xor.integer64), 25
- %%/, 4, 6, 26
- %%.integer64, 6
- %%/, 4, 6, 26
- %%.integer64, 6
- ^, 4, 6, 26
- ^.integer64, 6
- ^.integer64 (xor.integer64), 25

- abs, 6, 21
- abs.integer64, 6
- abs.integer64 (format.integer64), 20
- all, 7, 25
- all.integer64, 7
- all.integer64 (sum.integer64), 24
- any, 7, 25
- any.integer64, 7
- any.integer64 (sum.integer64), 24
- array, 8
- as.bitstring, 5
- as.bitstring (as.character.integer64), 14
- as.bitstring.integer64, 5
- as.character, 5, 14
- as.character.integer64, 5, 14, 16
- as.data.frame, 5, 15
- as.data.frame.integer64, 5, 15, 17, 23, 24
- as.double, 4, 5, 14
- as.double.integer64, 5
- as.double.integer64 (as.character.integer64), 14
- as.factor, 9
- as.integer, 5, 14
- as.integer.integer64, 5
- as.integer.integer64 (as.character.integer64), 14
- as.integer64, 5
- as.integer64 (as.integer64.character), 16
- as.integer64.character, 5, 14, 16
- as.integer64.double, 5
- as.integer64.integer, 5
- as.integer64.integer64, 5

- as.integer64.logical, 5
- as.integer64.NULL, 5
- as.logical, 5, 14
- as.logical.integer64, 5
- as.logical.integer64 (as.character.integer64), 14
- as.vector, 5, 8
- as.vector.integer64, 5, 15
- as.vector.integer64 (bit64-package), 2
- atomic, 3
- attr<-, 4
- attributes<-, 4

- binattr, 7
- binattr (xor.integer64), 25
- bit, 4, 8
- bit64 (bit64-package), 2
- bit64-package, 2
- bitwhich, 8

- c, 4, 5, 8, 17
- c.integer64, 5, 8, 17, 23, 24
- cbind, 4, 5, 17
- cbind.integer64, 5, 15
- cbind.integer64 (c.integer64), 17
- ceiling, 6, 21
- ceiling.integer64, 6
- ceiling.integer64 (format.integer64), 20
- character, 5
- cummax, 6, 18
- cummax.integer64, 6
- cummax.integer64 (cumsum.integer64), 18
- cummin, 6, 18
- cummin.integer64, 6
- cummin.integer64 (cumsum.integer64), 18
- cumprod, 6, 18
- cumprod.integer64, 6
- cumprod.integer64 (cumsum.integer64), 18
- cumsum, 6, 18
- cumsum.integer64, 6, 18, 25

- data.frame, 5
- dget, 7
- diff, 6, 18
- diff.integer64, 6
- diff.integer64 (cumsum.integer64), 18
- dim, 3, 5
- dim<-, 5
- dim<-, 3

- dimnames, 3, 5
- dimnames<-, 5
- double, 3–5, 7
- dput, 7
- duplicated, 9
- extract.replace.integer64, 19
- ff, 4, 7
- floor, 6, 21
- floor.integer64, 6
- floor.integer64 (format.integer64), 20
- format, 6, 14, 21
- format.integer64, 6, 20, 26
- identical, 5, 7, 21, 22
- identical.integer64, 5, 7, 21
- inherits, 3
- int64, 3, 9
- integer, 3–5, 9
- integer64, 14–19, 21–26
- integer64 (bit64-package), 2
- is, 4
- is.atomic, 3
- is.double, 4, 8
- is.integer, 5
- is.integer.integer64 (bit64-package), 2
- is.integer64, 4
- is.integer64 (bit64-package), 2
- is.na, 6, 21
- is.na.integer64, 6
- is.na.integer64 (format.integer64), 20
- is.vector, 4, 5, 8
- is.vector.integer64, 5, 8
- is.vector.integer64 (bit64-package), 2
- length, 3, 5
- length<-, 5
- length<- .integer64, 5
- length<- .integer64 (bit64-package), 2
- length<- , 3
- lim.integer64 (sum.integer64), 24
- load, 7
- log, 4, 6, 21
- log.integer64, 6
- log.integer64 (format.integer64), 20
- log10, 4, 6, 21
- log10.integer64, 6
- log10.integer64 (format.integer64), 20
- log2, 4, 6, 21
- log2.integer64, 6
- log2.integer64 (format.integer64), 20
- logical, 5
- match, 9
- matrix, 8
- max, 6, 25
- max.integer64, 6
- max.integer64 (sum.integer64), 24
- min, 6, 25
- min.integer64, 6
- min.integer64 (sum.integer64), 24
- minusclass, 7
- mode, 4
- mode<-, 8
- names, 3, 5
- names<-, 5
- names<- , 3
- NextMethod, 3, 14, 16, 17, 19, 21, 22
- NULL, 5
- order, 9
- plusclass, 7
- print, 5
- print.integer64, 5
- print.integer64 (bit64-package), 2
- prod, 6, 25
- prod.integer64, 6
- prod.integer64 (sum.integer64), 24
- range, 4, 6, 25
- range.integer64, 6
- range.integer64 (sum.integer64), 24
- rbind, 4, 5, 17
- rbind.integer64, 5
- rbind.integer64 (c.integer64), 17
- read.table, 7
- rep, 4, 22
- rep.integer64, 4, 17, 22, 23, 24
- round, 6, 21
- round.integer64, 6
- round.integer64 (format.integer64), 20
- S3, 3
- S4, 3
- save, 7
- seq, 4, 23

seq.integer64, 4, 17, 23
serialize, 3, 7
setattr, 4
setattributes, 4
sign, 6, 21
sign.integer64, 6
sign.integer64 (format.integer64), 20
signif, 6, 21
signif.integer64, 6
signif.integer64 (format.integer64), 20
sort, 9
sqrt, 4, 6, 21
sqrt.integer64, 6
sqrt.integer64 (format.integer64), 20
storage.mode, 4
storage.mode<-, 8
str, 5, 8
sum, 6, 25
sum.integer64, 6, 18, 24

table, 9
trunc, 6, 21
trunc.integer64, 6
trunc.integer64 (format.integer64), 20
typeof, 4

unique, 9
unlist, 8
unserialize, 7

vector, 8
vmode, 4

warning, 25
write.table, 7

xor, 6, 26
xor.integer64, 6, 21, 25