CHAPTER 8

Interfacing with C and C++

There are several levels at which the ICI interpreter can interface with C and C++ code. This chapter gives a collection of universal rules, then addresses common tasks. Each task can be considered in isolation to alleviate the reader from details beyond their current needs. Finally, a summary of ICI C API elements is given.

Some functions and macros are documented best in the comments in the ICI source code. The reader is expected to be a C/C++ programmer who is able to find and read such comments when this document gives only a general indication of function and example usage.

Universal rules and conventions

Include files and libraries

On most systems ICI is built as a dynamically loading library. It can be linked with statically if required. However for this description I will assume the normal case of dynamic loading.

To compile and run with ICI you will need, as a minimum:

ici.h	The ICI include file. Include this as necessary. This file is built specifically for each platform from ICI's internal include files.
icistr-setup.h	A utility include file to assist in defining ICI strings. See *Referring to ICI strings from C* below.
ici4.{a,lib}	The library file containg the linkage to the dynamically loading library (or the static library if not using dynamic loading). (Suffixes vary with OS.) Link against this when building your program.
ici4.{so,dll}	The dynamic loading library. (Suffixes vary with OS.) Should be somewhere where it will be found at run time.

ici4core.ici, ici4core{1,2,3}.ici Core language features written in ICI. These need to be somewhere ICI will find them at run time.

If you are writing modules that run from ICI, you will also want an ICI top level command:

ici, ici.exe, or *wici.exe* ICI command level executable. These just do argument parsing and invocation of the interpreter on supplied file arguments.

In broad terms ICI is either used as an adjunct to another application, or as the main program with specific custom modules providing special functionality. Most of what is described in this section is common to both.

The nature of ICI objects

ICI objects are structures that have a common 32 bit header (since version 4; in previous versions it was a 64 bit header). Pointers to objects are declared as either *object_t **, which is the type of the header, or a pointer to the particular structure (for example, *string_t **), depending on the circumstances of each piece of code, and depending whether the real type is known at that point. The macro *objof()* is often used to demote to a generic object pointer -- it is just a cast to *object_t **. Most types define a similar macro to promote to their specific type, as well as a macro such as *isstring()* or *isstruct()* to test if a particular pointer points to an object of the given type. However, there is no particular requirement to use these macros. They are just casts and simple tests.

Garbage collection, *ici_incref()* and *ici_decref()*

ICI objects are garbage collected. Garbage collection can occur on any attempt to allocate memory using ICI's allocation routines. This is fairly often. Garbage collection has the effect of freeing any objects that the garbage collector thinks are not being usefully referenced. Failing to obey the rules associated with garbage collection can be disastrous and hard to debug. But the rules are fairly simple.

The ICI object header includes a *small* reference count field. This is a count of additional references to the object that would otherwise be invisible to the garbage collector. For example, if your C code keeps a global static pointer to an object, the garbage collector would not be aware of that reference, and might free the object leaving you with an invalid pointer. So you must account for the reference by incrementing the reference count. However, references to the object from other ICI objects are visible to the garbage collector, so you do not need to account for these.

The macros *ici_incref()* and *ici_decref()* are used to increment and decrement reference counts. Note that the range of reference counts is quite small. Their frequency of use is expected to be in relationship to the number of actual instances of C variables that simultaneously refer to the same object.

In practice, many calls to *ici_incref()* and *ici_decref()* can be avoided because objects are known to be referenced by other things. For example, when coding a function called from ICI, the objects that are passed as arguments are known to be referenced from the ICI operand stack, which is referenced by the current execution context, which has a reference count as long as it is running.

The error return convention

When coding functions that are called by the ICI interpreter, a simple mechanism is used for all error returns. Each function will have some return value to indicate an error (commonly 0 is success, 1 is an error, or for functions that return a pointer, NULL will probably be the error indicator). In any case, the return value will only imply a boolean indicating that an error has occurred. The nature of the error *must* have been set before return by setting the global character pointer *ici_error* to a short human readable string revealing what happened.

Note, however, that *only* the originator of an error condition should set *ici_error*. If you call another function that uses the error return convention, and it returns a failure, you must simply clean up your immediate situation (such as any required *ici_decref()* calls) and return your failure indication in turn. For example, the *ici_alloc()* function obeys the convention. Thus we might see the fragment:

```
if ((p = ici_alloc(sizeof(mytype)) == NULL)
     return 1;
```

Now, in setting and returning an error, your code will be losing control and must be concerned about the memory that you set the *ici_error* variable to point to. Simple static strings are, of course, of no concern. For example:

```
if (that_failed)
{
    ici_error = "failed to do that";
    return 1;
}
```

If you need to format a string (say with sprintf) you can avoid the necessity of a permanently allocated buffer by using a generic growable buffer provided by ICI. The buffer is pointed to by *ici_buf* and *always* has room for at least 120 characters. Thus me might see:

```
if (v > 256)
{
    sprintf(ici_buf, "%d set but 256 is limit", v);
    ici_error = ici_buf;
}
```

If the size of the generated message is not so limited, the buffer can be checked (or forced) to have a larger size with *ici_chkbuf()*. For example:

```
if (file == NULL)
{
    if (ici_chkbuf(40 + strlen(fname))
        return 1;
    sprintf(ici_buf, "could not open %s", fname);
    ici_error = ici_buf;
    return 1;
}
```

Notice *ici_chkbuf()* could fail, and if so, we return immediately (the error will inevitably be “ran out of memory”). The *40* above is some number clearly larger than the length of the format string.

One final point, which is not specifically to do with error returns, but commonly associated with them, is how to get a short human readable description of an ICI object suitable for diagnostics. The function *ici_objname()* can be used to get a small (guaranteed to fit within a 30 character buffer) diagnostic description of an object. For example:

```
{
    char        n1[30];
    char        n2[30];

    sprintf(ici_buf, "attempt to read %s keyed by %s",
        ici_objname(n1, o),
        ici_objname(n2, k));
    ici_error = ici_buf;
    return 1;
}
```

ICI's allocation functions

In general ICI uses the native *malloc()* and *free()* functions for its memory allocation. However, because ICI deals in many small objects, and because it needs to track memory usage to control when to run its mark/sweep garbage collector, ICI has a few allocation functions that you may wish to be aware of.

There are three pairs of matched alloc/free functions. Memory allocated with one must be freed by the matching one. Common to all three is the property that they (try) to track the amount of memory allocated so that they can trigger garbage collections, and they may run the garbage collector before they return.

Two of them (*ici_talloc* / *ici_tfree* and *ici_nalloc* / *ici_nfree*) are designed to handle small objects efficiently (small meaning up to 64 bytes). They allocate small objects densely (no boundary words) out of larger raw allocations from *malloc()* and maintain fast free lists so that most allocations and frees can avoid function calls completely. However, in order to avoid boundary words they both require that the caller tells the free routines how much memory was asked for on the initial alloc. This is fairly easy 95% of the time, but where it can't be managed, you must use the simpler *ici_alloc* / *ici_free* pair. These are completely *malloc()* equivalent, except they handle the garbage collection and have the usual ICI error return convention.

The tracking of memory usage is only relevant to memory the garbage collector has some control over, meaning memory associated with ICI objects (that would get freed if the object was collected). So, technically, these routines should be used for memory associated with objects, and not other allocations. But in practice the division is not strict.

Common tasks

Writing a simple function that can be called from ICI

This is sometimes done as part of a dynamically loaded extension module, and at other times done in a program that uses ICI as a sub-module. By *simple function* we mean a function that takes and returns values that are directly equivalent to simple C data types.

Transfer from the ICI execution engine to an "intrinsic" function (as they are called) is designed to have extremely low overhead. Thus, on arrival at a function, the arguments are still ICI objects on the ICI operand stack. If you are dealing with simple argument and return types, you can then use the *ici_typecheck()* function to marshal your arguments into C variables, and a set of similar functions to return simple values.

Intrinsic functions have a return type of *int*, use C calling convention, follow the usual error convention (1 is failure) and in simple cases declare no arguments. For example, a function that expects to be called with an *int* and a *float* from ICI, print them, and return the int plus one, would read:

```
static int
f_myfunc()
{
    long          i;
    double        f;

    if (ici_typecheck("if", &i, &f))
        return 1;
    printf("Got %ld, %lf.\n", i, f);
    return int_ret(i + 1);
}
```

Note that ICI ints are C longs, and ICI floats are C doubles.

The *ici_typecheck()* function allows marshalling of:

- ICI ints to C longs, floats to doubles, and strings to char pointers;
- generic "numeric" (int or float) values to a C double;
- many other ICI types (structs, arrays, generic objects, etc) to generic object pointers (see other tasks on how to deal with them after that);
- ICI pointers to any of the above.

It also supports skipping argument and variable argument lists, but these features are typically used in functions that use a mixture of *ici_typecheck()* calls and explicit argument processing. See *ici_typecheck()* in *cfunc.c*.

To return simple data types, the functions *ici_int_ret()*, *ici_float_ret()*, *ici_str_ret()*, and *ici_null_ret()* can be used. These convert the value to the approriate ICI data object, and replace the arguments on the operand stack with that object, then return 0.

Take care *never* to simply *"return 0;"* from an intrinsic function. Although returning 1 on error is correct, and the non-error return value is zero, before returning the arguments must be replaced with the return value on the ICI operand stack. The functions above, and various others, do this. They should always be used on successful return from any intrinsic function.

It is possible to write a function that is passed pointers to values from ICI, and have those values updated by the intrinsic function, by using a combination of the *ici_typecheck()* function and the *ici_retcheck()* function. For example:

```
static int
f_myotherfunc()
{
    long          i;
```

```
    double        f;

    if (ici_typecheck("fI", &f, &i))
        return 1;
    printf("Got %ld, %lf.\n", i, f);
    if (ici_retcheck("-i", i + 1))
        return 1;
    return float_ret(f * 3.0);
}
```

This function takes a float, and a pointer to an int. It returns three times the passed float value, and increments the pointed-to int. Notice the capitalisation in the *ici_typecheck()* call to indicate a pointer to an int is required. Also note the hyphen in the *ici_retcheck()* call to indicate it should ignore the first argument.

Calling an ICI function from C

Calling an ICI function from C is fairly simple. There are a few slight variations on the same basic call. The simplest is the

Making new ICI primitive objects

Writing and compiling a dynamically loading extension module

The loaded library must contain a function of the following name and declaration:

```
object_t *
ici_var_library_init()
{
    ...
}
```

where *var* is the as yet undefined variable name. This is the initialisation function which is called when the library is loaded. This function should return an ICI object, or NULL on error, in which case the ICI error variable must be set. The returned object will be assigned to *var* as described above.

The following sample module, *mbox.c*, illistrates a typical form for a simple dynamically loaded ICI module (it is a Windows example, but should be clear anyway):

```
#include <windows.h>
#include <ici.h>

/*
 * mbox_msg => mbox.msg(string) from ICI
 *
 * Pops up a modal message box with the given string in it
 * and waits for the user to hit OK. Returns NULL.
 */
int
mbox_msg()
{
```

```
    char    *msg;

    if (typecheck("s", &msg))
        return 1;
    MessageBox(NULL, msg, (LPCTSTR)"ICI", MB_OK | MB_SETFOREGROUND);
    return ici_null_ret();
}

/*
 * Object stubs for our intrinsic functions.
 */
cfunc_t mbox_cfuncs[] =
{
    {CF_OBJ,  "msg",  mbox_msg},
    {CF_OBJ}
};

/*
 * ici_mbox_library_init
 *
 * Initialisation routine called on load of this module into the ICI
 * interpreter's address space. Creates and returns a struct which will
 * be assigned to "mbox". This struct contains references to our
 * intrinsic functions.
 */
object_t *
ici_mbox_library_init()
{
    objwsup_t        *s;

    if ((s = ici_module_new(mbox_cfuncs)) == NULL)
        return NULL;
    return objof(s);
}
```

The following simple Makefile illustrates forms suitable for compiling this module into a DLL under Windows. Note in particular the use of /export in the link line to make the function *ici_mbox_library_init* externally visible.

```
CFLAGS=
OBJS  = mbox.obj
LIBS  = ici4.lib user32.lib

icimbox.dll: $(OBJS)
            link /dll /out:$@ $(OBJS) /export:ici_mbox_library_init $(LIBS)
```

Note that there is no direct supprt for the /export option in the MS Developer Studio link settings panel, but it can be entered directly in the *Project Options* text box.

The following Makefile achieves an equivalent result under Solaris:

```
CC    = gcc -pipe -g
CFLAGS= -fpic -I..

OBJS  = mbox.o

icimbox.so  : $(OBJS)
            ld -o $@ -dc -dp $(OBJS)
```

Referring to ICI strings from C code

References to short strings that are known at compile time is common in ICI modules for field names and such-like. But ICI strings need to be looked up in the ICI atom pool to find the unique pointer for each particular one (and created if it does not already exist). To assist external modules in obtaining the pointer to names they need (especially when there are lots), some macros are defined in *ici.h* to assist. The following procedure can be used:

1. Make an include file called *icistr.h* with your strings, and what you want to call them, formatted as in this example:

```
/*
 * Any strings listed in this file may be referred to
 * with ICIS(name) for a (string_t *), and ICISO(name)
 * for an (object_t *).
 *
 * This file is included with varying definitions
 * of ICI_STR() to declare, define, and initialise
 * the strings.
 */
ICI_STR(fred, "fred")
ICI_STR(jane, "jane")
ICI_STR(amp, "&")
```

2. Next, in one of your source files, after an include of *ici.h*, include the special include file *icistr-setup.h*. That is:

```
#include <icistr-setup.h>
```

   This include file both defines variables to hold pointer to the strings (based on the names you gave in *icistr.h*) and defines a function called *init_ici_str()* which initialises those pointers. It does this by including your *icistr.h* file twice, but under the influence of special defines for *ICI_STR()*.

3. Next, call *init_ici_str()* at startup or library load. It returns 1 on error, usual conventions. For example:

```
object_t *
ici_XXX_library_init(void)
{
    if (init_ici_str())
        return NULL;
    ...
```

4. Include your *icistr.h* file in any source files that accesses the named ICI strings. Access them with either *ICIS(fred)* or *ICISO(fred)* which return *string_t* * and *object_t** pointers respectively. For example:

```
#include "ici.h"
#include "icistr.h"
...
    object_t     *o;
    struct_t     *s
...
```

```
o = ici_fetch(s, ICIS(fred));
```

Accessing ICI array objects from C

Using ICI independently from multiple threads

Summary of ICI's C API

The following table summarises function and public data that form ICI's C API. The use of some of these functions has been illustrated above. The full specification of each is given in a comment in the ICI source. This summary is only intended to document the limits of the public interface, allow you to select the relevant functions, and direct you to the source with the full description.

The final column gives a hint about upgrading from ICI 3. If just a name is mentioned, that is the old name of this function (many names have acquired an *ici_* prefix). Other notes indicate what constructs should be checked for possible upgrade to the given function. If it is blank, there is no change. There is an ICI program, *ici3check.ici*, in the ICI source directory that will grep your source for usage of changed constructs and print a note on upgrade action.

Name	Synopsis	Upgrade from ICI3
ici.h	The ICI C API include file. This is generated specifically for each platform.	
ici_alloc	Allocate memory, in particular memory subject to collection (possibly indirectly). Must be freed with *ici_free*. See also *ici_talloc* and *ici_nalloc* which are both preferable. See *alloc.c*.	
ici_argcount	Generate an error message indicating that this intrinsic function was given the wrong number of argument. See *cfunc.c*.	
ici_argerror	Generate an errror message indicating that this argument of this intrinsic function is wrong. See *cfunc.c*.	
ici_array_gather	Copy a (possibly disjoint) run of elements of an array into contiguous memory.	Use of a_top.
ici_array_get	Fetch an elemet of an array indexed by a C int. See *array.c*.	Use of a_top.
ici_array_nels	Return the number of elements in an array. See *array.c*.	Use of a_top.
ici_array_new	Allocate a new ICI array. See *array.c*.	new_array
ici_array_pop	Pop and return last element of an ICI array. See *array.c*.	Use of a_top.
ici_array_push	Push an object onto an array. See *array.c*.	Use of a_top.
ici_array_rpop	Rpop an obect from an ICI array. See *array.c*.	
ici_array_rpush	Rpush an oject onto an ICI array. See *array.c*.	
ici_array_t	The ICI array object type. See *array.h*.	array_t
ici_assign	Assign to an ICI object (vectors through type). See *object.h*.	assign
ici_assign_cfuncs	Assign of bunch of intrinsic function prototypes into the ICI namespace. See *cfunco.c*.	
ici_assign_fail	Generic function that can be used for types that don't support assignment. See *object.c*.	assign_simple
ici_atexit	Register a function to be called at *ici_uninit*. See *uninit.c*.	
ici_atom	Return the atomic form of an object. See *object.c*.	atom
ici_buf	A general purpose growable character buffer, typically used for error messages. See *ici_chkbuf*. See *buf.c*.	
ici_call	Call an ICI function from C by name. See *call.c*.	Prototype change.

Name	Synopsis	Upgrade from ICI3
ici_callv	Same as *ici_call* but takes a *va_list*. See *call.c*.	Prototype change.
ici_cfunc_t	The ICI intrinsic function object types. See *cfunc.h* and *cfunco.c* (not *cfunc.c*).	cfunc_t
ici_chkbuf	Verify or grow *ici_buf* to be big enough. See *buf.c*.	
ici_cmp_unique	Generic function that can be used for types that can't be merged through the atom pool. See *object.c*.	cmp_unique
ici_copy_simple	Generic copy function that can be used for types that are intrinsically atomic. See *object.c*.	copy_simple
ici_debug	A pointer to the current debug functions. See *idb2.c*.	
ici_debug_enabled	If compiled with debug, an int giving the current status of debug callbacks. Else defined to 0 by the preprocessor. See *fwd.h*.	
ici_debug_t	A struct of function pointers for debug functions (like break and watch). See *exec.h*.	debug_t
ici_decref	Decrement the reference count of an ICI object. See *object.h*.	decref
ici_def_cfunc	Define C functions in the current scope. See *cfunco.c*.	def_cfuncs
ici_dont_record_line_nums	A global int which can be set to prevent line number records (which will marginally speed execution, but errors won't reveal source location). See *fwd.h*.	
ici_enter	Acquire the global ICI mutex, which is required for access to ICI data and functions. See *thread.c*.	
ici_error	A global char pointer to any current error message.	
ici_exec	A pointer to the current ICI execution context (NB: Don't look at or touch x_os, x_xs or x_vs fields). See *exec.h*.	
ici_fetch	Fetch an element from an object. Vectors by object type. See *object.h*.	fetch
ici_fetch_fail	A generic function that can be used by objects that don't support fetching.	fetch_simple
ici_fetch_int	Fetch an int from an object into a C long. See *mkvar.c*.	
ici_fetch_num	Fetch an int or float from an object into a C double. See *mkvar.c*.	
ici_file_close	Close the low-level file associated with an ICI file object. See *file.c*.	file_close
ici_file_new	Create a new ICI file object. See *file.c*.	new_file

Name	Synopsis	Upgrade from ICI3
ici_file_t	The ICI file object type. See *file.h*.	file_t
ici_float_new	Get an ICI float from a C double. See *float.c*.	new_float
ici_float_ret	Return a C double from an intrinsic function as an ICI float. See *cfunc.c*.	float_ret
ici_float_t	The ICI float object type. See *float.h*.	float_t
ici_free	Free memory allocated with *ici_alloc*. See also *ici_tfree* and *ici_nfree*. See *alloc.c*.	
ici_func	Call an ICI function, given you have a *func_t **. See *ici_call* to call by name. See *call.c*.	Prototype change.
ici_func_t	The ICI function object type. See *func.h*.	func_t
ici_funcv	Same as *ici_func* except it takes a *va_list*.	Prototype change.
ici_get_last_errno	Set *ici_error* (see) based on the last failed system function (i.e. *errno*). See *syserr.c*.	syserr (and semantics change)
ici_get_last_win32_error	Windows only. Set *ici_error* based on the value of *GetLastError()*. See *win32err.c*.	
ici_handle_new	Make a new *handle_t* object.	
ici_handle_t	The type of handle objects. See *handle.c*.	
ici_hash_unique	A generic function that can be used in a *type_t* struct for objects that can't be merged through the atom pool. See *object.c*.	
ici_incref	Increment reference to an ICI object.	incref
ici_init	Initialise the ICI interpreter. See *init.c*.	
ici_int_new	Get an ICI int from a C long. See *int.c*.	new_int
ici_int_ret	Return a C long from an intrinsic function.See *cfunc.c*.	int_ret
ici_int_t		int_t
ici_leave	Unlock the global ICI mutex to allow thread switching. See *thread.c*.	
ici_main	A wrapper round *ici_init()* that does argc, argv argument processing. See *icimain.c*.	
ici_mark	Mark an object as part of the garbage collection mark phase. See *object.h*.	mark
ici_mem_new	Allocate a new *mem_t* object. See *mem.c*.	new_mem
ici_mem_t		mem_t
ici_method	Call an ICI method given an instance and a method name. See *call.c*.	
ici_method_new	Allocate a new *metnod_t* object. See *method.c*.	ici_new_method
ici_method_t		method_t
ici_nalloc	Allocate memory, in particular memory subject to collection (possibly indirectly). Must be freed with *ici_nfree*. See also *ici_talloc* and *ici_alloc*. See *alloc.c*.	

Name	Synopsis	Upgrade from ICI3
ici_need_stdin	Return ICI file object that is the current value of stdin.	need_stdin
ici_need_stdout	Return ICI file object that is the current value of stdout.	need_stdout
ici_nfree	Free memory allocated with *ici_nalloc*. See also *ici_tfree* and *ici_free*. See *alloc.c*.	
ici_null	A pointer to the ICI NULL object.	objof(&o_null)
ici_null_ret	Return an ICI NULL from an intrinsic function.See *cfunc.c*.	null_ret
ici_null_t		null_t
ici_obj_t	The generic ICI object type. All ICI objects have this as their first element.	object_t
ici_objname	Get a short human readable representation of any object for diagnostics reports. See *cfunc.c*.	objname
ici_objwsup_t		objwsup_t
ici_one	A global pointer to the ICI int 1.	o_one
ici_os	An ICI array which is the operand stack of the current execution context.	
ici_parse_file	Parse a file as a new top-level module.	parse_file
ici_ptr_new	Allocate a new ICI *ptr* object. See *ptr.c*.	new_ptr
ici_ptr_t		ptr_t
ici_reclaim	Run the ICI garbage collector. See *object.c*.	
ici_regexp_new	Make a new ICI regexp object.	new_regexp
ici_regexp_t		regexp_t
ici_register_type	Register a new *type_t* structure with the interpreter to obtain the small int type code that must be placed in the header of ICI objects.	o_type assignment
ici_rego	Register a new object with the garbage collector. See *object.h* (a macro).	rego
ici_ret_no_decref	Return an ICI object from an intrinsic C function, without an *ici_decref*. See *cfunc.c*.	
ici_ret_with_decref	Return an ICI object from an intrinsic C function, but *ici_decref* it in the process. See *cfunc.c*.	
ici_retcheck	Check and update values returned through pointers.	
ici_set_new	Allocate a new ICI set object. See *set.c*.	new_set
ici_set_t		set_t
ici_set_unassign	Remove an element from a set. See *set.c*.	unassign_set
ici_set_val	Set a C int, long, double, FILE * or ICI object into the inner-most scope of any object that supports a super. See *mkvar.c*.	

Name	Synopsis	Upgrade from ICI3
ici_skt_t		skt_t
ici_stdio_ftype	A struct holding pointers to stdio functions to facilitate making stdio based ICI files.	stdio_ftype
ici_stk_push_chk	Ensures there is a certain amount of contiguous room at the end of an array for direct push operations through a_top. See *array.h*.	
ICI_STR	Multi-purpose macro used by the *icistr-setup.h* mechanism. See *str.h*.	
ici_str_alloc	Do half of the allocation of a string. Data must be added and *ici_atom* called before completion. See *string.c*.	new_string
ici_str_get_nul_term	Get the ICI form of a string of nul terminated chars without an extra reference count. See *string.c*.	get_cname
ici_str_new	Get the ICI form of a string of chars, by explicit length. See *string.c*.	new_name
ici_str_new_nul_term	Get the ICI form of a string of nul terminated chars. See *string.c*.	new_cname
ici_str_ret	Return a nul terminated C string from an intrinsic function as an ICI string. See *cfunc.c*.	str_ret
ici_str_t		string_t
ici_struct_new	Allocate a new ICI struct object. See *struct.c*.	new_struct
ici_struct_t		struct_t
ici_struct_unassign	Remove an element from an ICI struct. See *struct.c*.	unassign_struct
ici_talloc	Allocate memory, in particular memory subject to collection (possibly indirectly) sufficient for a given type. Must be freed with *ici_tfree*. See also *ici_nalloc* and *ici_alloc*. See *alloc.c*.	ici_alloc
ici_tfree	Free memory allocated with *ici_tfree*. See *alloc.c*.	
ici_type_t	The type that holds information about ICI primitive object types. You must declare, initialise, and register one of these to make a new ICI primitive type. See *ici_register_type*. See *object.h*.	type_t
ici_typecheck	Check and marshall ICI arguments to an intrinsic function into C variables. See *cfunc.c*.	

Name	Synopsis	Upgrade from ICI3
ici_uninit	Shutdown and free resources associated with the ICI interpreter. See *uninit.c*.	
ici_vs	The scope ("variable") stack of the current ICI execution context.	
ici_wakeup	Wake up any ICI threads waiting on a given ICI object. See *thread.c*.	
ici_wrap_t	Struct to support *ici_atexit*. See *uninit.c*.	wrap_t
ici_xs	The execution stack of the current ICI execution engine. See *exec.c*.	
ici_*XXX*_library_init	The entry point you must define for dynamically loaded modules.	
ici_zero	The ICI int 0.	o_zero
icistr.h	The include file you must make to get initialised ICI strings. See *str.h*.	
icistr-setup.h	The include file you must include to initialised ICI strings. See *str.h*.	
objof	Macro to cast an object to an (object_t *)	

Detailed description of ICI's C API

ICI_NO_OLD_NAMES

```
#define ICI_NO_OLD_NAMES  ...
```

This define may be made before an include of *ici.h* to suppress a group of old (backward compatible) names. These names have been upgraded to have *ici_* prefixes since version 4.0.4. These names don't effect the binary interface of the API; they are all type or macro names. But you might want to suppress them if you get a clash with some other include file (for example, *file_t* has been known to clash with defines in <*file.h*> on some systems).

If you just was to get rid of one or two defines, you can *#undef* them after the include of *ici.h*.

The names this define supresses are:

```
array_t     float_t     object_t    catch_t
slot_t      set_t       struct_t    exec_t
file_t      func_t      cfunc_t     method_t
int_t       mark_t      null_t      objwsup_t
op_t        pc_t        ptr_t       regexp_t
src_t       string_t    type_t      wrap_t
ftype_t     forall_t    parse_t     mem_t
debug_t
```

ici_alloc

```
void * ici_alloc(size_t z)
```

Allocate a block of size *z*. This just maps to a raw malloc() but does garbage collection as necessary and attempts to track memory usage to control when the garbage collector runs. Blocks allocated with this must be freed with ici_free().

It is preferable to use ici_talloc(), or failing that, ici_nalloc(), instead of this function. But both require that you can match the allocation by calling ici_tfree() or ici_nalloc() with the original type/size you passed in the allocation call. Those functions use dense fast free lists for small objects, and track memory usage better.

See also: *ICIs allocation functions*, ici_free(), ici_talloc(), ici_nalloc().

ici_argcount

```
int ici_argcount(int n)
```

Generate a generic error message to indicate that the wrong number of arguments have been supplied to an intrinsic function, and that it really (or most commonly) takes *n*. This function sets the error descriptor (ici_error) to a message like:

```
%d arguments given to %s, but it takes %d
```

and then returns 1.

This function may only be called from the implementation of an intrinsic function. It takes the number of actual argument and the function name from the current operand stack, which therefore should not have been distured (which is normal for intrincic functions). It takes the number of arguments the function should have been supplied with (or typically is) from *n*. This function is typically used from C coded functions that are not using ici_typecheck() to process arguments. For example, a function that just takes a single object as an argument might start:

```
static int
myfunc()
{
    ici_obj_t  *o;

    if (NARGS() != 1)
        return ici_argcount(1);
    o = ARG(0);
    . . .
```

ici_argerror

```
int ici_argerror(int i)
```

Generate a generic error message to indicate that argument i of the current intrinsic function is bad. Despite being generic, this message is generally pretty informative and useful. It has the form:

```
argument %d of %s incorrectly supplied as %s
```

The argument number is base 0. I.e. ici_argerror(0) indicates the 1st argument is bad.

The function returns 1, for use in a direct return from an intrinsic function.

This function may only be called from the implementation of an intrinsic function. It takes the function name from the current operand stack, which therefore should not have been distured (which is normal for intrincic functions). This function is typically used from C coded functions that are not using ici_typecheck() to process arguments. For example, a function that just takes a single mem object as an argument might start:

```
static int
myfunc()
{
    ici_obj_t  *o;

    if (NARGS() != 1)
        return ici_argcount(1);
    if (!ismem(ARG(0)))
        return ici_argerror(0);
    . . .
```

ici_array_gather

```
void ici_array_gather(ici_obj_t **b, ici_array_t *a,
ptrdiff_t start, ptrdiff_t n)
```

Copy *n* object pointers from the given array, starting at index *start*, to *b*. The span must cover existing elements of the array (that is, don't try to read from negative or excessive indexes).

This function is used to copy objects out of an array into a contiguous destination area. You can't easily just memcpy, because the span of elements you want may wrap around the end. For example, the implementaion of interval() uses this to copy the span of elements it wants into a new array.

ici_array_get

```
ici_obj_t * ici_array_get(ici_array_t *a, ptrdiff_t i)
```

Return the element or the array *a* from index *i*, or *ici_null* if out of range. No incref is done on the object.

ici_array_nels

```
ptrdiff_t ici_array_nels(ici_array_t *a)
```

Return the number of elements in the array *a*.

ici_array_new

```
ici_array_t * ici_array_new(ptrdiff_t n)
```

Return a new array. It will have room for at least *n* elements to be pushed contigously (that is, there is no need to use ici_stk_push_chk() for objects pushed immediately, up to that limit). If *n* is 0 an internal default will be used. The returned array has ref count 1. Returns NULL on failure, usual conventions.

ici_array_new

```
ici_array_t * ici_array_new(ptrdiff_t n)
```

Return a new array. It will have room for at least *n* elements to be pushed contigously (that is, there is no need to use ici_stk_push_chk() for objects pushed immediately, up to that limit). If *n* is 0 an internal default will be used. The returned array has ref count 1. Returns NULL on failure, usual conventions.

ici_array_pop

```
ici_obj_t * ici_array_pop(ici_array_t *a)
```

Pop and return the top of the given array, or *ici_null* if it is empty. Returns NULL on error (for example, attempting to pop and atomic array). Usual error conventions.

ici_array_push

```
int ici_array_push(ici_array_t *a, ici_obj_t *o)
```

Push the object *o* onto the end of the array *a*. This is the general case that works for any array whether it is a stack or a queue. On return, o_top[-1] is the object pushed. Returns 1 on error, else 0, usual error conventions.

ici_array_rpop

```
ici_obj_t * ici_array_rpop(ici_array_t *a)
```

Pop and return the front of the given array, or *ici_null* if it is empty. Returns NULL on error (for example, attempting to pop and atomic array). Usual error conventions.

ici_array_rpush

```
int ici_array_rpush(ici_array_t *a, ici_obj_t *o)
```

Push the object *o* onto the front of the array *a*. Return 1 on failure, else 0, usual error conventions.

ici_assign_cfuncs

```
int ici_assign_cfuncs(ici_objwsup_t *s, ici_cfunc_t *cf)
```

Assign the structure s all the intrinsic functions listed in the array of ici_cfunc_t structures pointed to by cf. The array must be terminated by an entry with a cf_name of NULL. Typically, entries in the array are formated as:

```
{CF_OBJ,    "func",     f_func},
```

Where CF_OBJ is a convenience macro from to take care of the normal object header, "func" is the name your function will be assigned to in the given struct, and *f_func* is a C function obeying the rules of ICI intrinsic functions.

Returns non-zero on error, in which case error is set, else zero.

ici_call

```
int ici_call(ici_str_t *func_name, char *types, ...)
```

Call an ICI function by name from C with simple argument types and return value. The name (*func_name*) is looked up in the current scope.

See *ici_func()* for an explanation of *types*. Apart from taking a name, rather than an ICI function object, this function behaves in the same manner as *ici_func()*.

There is some historical support for @ operators, but it is deprecated and may be removed in future versions.

ici_call

```
int ici_call(ici_str_t *func_name, char *types, ...)
```

Call an ICI function by name from C with simple argument types and return value. The name (*func_name*) is looked up in the current scope.

See *ici_func()* for an explanation of *types*. Apart from taking a name, rather than an ICI function object, this function behaves in the same manner as *ici_func()*.

There is some historical support for @ operators, but it is deprecated and may be removed in future versions.

ici_callv

```
int ici_callv(ici_str_t *func_name, char *types, va_list
va)
```

Varient of *ici_call()* (see) taking a variable argument list.

There is some historical support for @ operators, but it is deprecated and may be removed in future versions.

ici_chkbuf

```
#define ici_chkbuf(n)  ...
```

Ensure that *ici_buf* points to enough memory to hold index *n* (plus room for a nul char at the end). Returns 0 on success, else 1 and sets *ici_error*.

See also: *The error return convention*.

ici_chkbuf

```
#define ici_chkbuf(n)  ...
```

Ensure that *ici_buf* points to enough memory to hold index *n* (plus room for a nul char at the end). Returns 0 on success, else 1 and sets *ici_error*.

See also: *The error return convention*.

ici_class_new

```
ici_objwsup_t * ici_class_new(ici_cfunc_t *cf,
ici_objwsup_t *super)
```

Create a new class struct and assign the given cfuncs into it (as in ici_assign_cfuncs()). If *super* is NULL, the super of the new struct is set to the outer-most writeable struct in the current scope. Thus this is a new top-level class (not derived from anything). If super is non-NULL, it is presumably the parent class and is used directly as the super. Returns NULL on error, usual conventions. The returned struct has an incref the caller owns.

ici_def_cfuncs

```
int ici_def_cfuncs(ici_cfunc_t *cf)
```

Define the given intrinsic functions in the current static scope. See ici_assign_cfuncs() for details.

Returns non-zero on error, in which case error is set, else zero.

ici_float_ret

```
int ici_float_ret(double ret)
```

Use *return ici_float_ret(n);* to return a float from an intrinsic fuction.

ici_free

```
void ici_free(void *p)
```

Free a block allocated with ici_alloc().

See also: *ICIs allocation functions*, ici_alloc(), ici_tfree(), ici_nfree().

ici_free

```
void ici_free(void *p)
```

Free a block allocated with ici_alloc().

See also: *ICIs allocation functions*, ici_alloc(), ici_tfree(), ici_nfree().

ici_func

```
int ici_func(ici_obj_t *callable, char *types, ...)
```

Call a callable ICI object *callable* from C with simple argument marshalling and an optional return value. The callable object is typically a function (but not a function name, see *ici_call* for that case).

types is a string that indicates what C values are being supplied as arguments. It can be of the form ".=..." or "..." where the dots represent type key letters as described below. In the first case

the 1st extra argument is used as a pointer to store the return value through. In the second case, the return value of the ICI function is not provided.

Type key letters are:

i — The corresponding argument should be a C long (a pointer to a long in the case of a return value). It will be converted to an ICI *int* and passed to the function.

f — The corresponding argument should be a C double. (a pointer to a double in the case of a return value). It will be converted to an ICI *float* and passed to the function.

s — The corresponding argument should be a nul terminated string (a pointer to a char * in the case of a return value). It will be converted to an ICI *string* and passed to the function.

When a string is returned it is a pointer to the character data of an internal ICI string object. It will only remain valid until the next call to any ICI function.

o — The corresponding argument should be a pointer to an ICI object (a pointer to an object in the case of a return value). It will be passed directly to the ICI function.

When an object is returned it has been ici_incref()ed (that is, it is held against garbage collection).

Returns 0 on success, else 1, in which case ici_error has been set.

See also: ici_callv(), ici_method(), ici_call(), ici_funcv().

ici_funcv

```
int ici_funcv(ici_obj_t *subject, ici_obj_t *callable, char
*types, va_list va)
```

This function is a variation on *ici_func()*. See that function for details on the meaning of the *types* argument.

va is a va_list (variable argument list) passed from an outer var-args function.

If *subject* is NULL, then *callable* is taken to be a callable object (could be a function, a method, or something else) and is called directly. If *subject* is non-NULL, it is taken to be an instance object and *callable* should be the name of one of its methods (i.e. an *ici_str_t* *).

ici_incref

```
#define ici_incref(o)  ...
```

References from ordinary machine data objects (ie. variables and stuff, not other objects) are invisible to the garbage collector. These refs must be accounted for if there is a possibility of garbage collection. Note that most routines that make objects (new_*(), copy() etc...) return

objects with 1 ref. The caller is expected to ici_decref() it when they attach it into wherever it is going.

ici_init

```
int ici_init(void)
```

Perform basic interpreter setup. Return non-zero on failure, usual conventions.

After calling this the scope stack has a struct for autos on it, and the super of that is for statics. That struct for statics is where global definitions that are likely to be visible to all code tend to get set. All the intrinsic functions for example. It forms the extern scope of any files parsed at the top level.

In systems supporting threads, on exit, the global ICI mutex has been acquired (with ici_enter()).

ici_int_ret

```
int ici_int_ret(long ret)
```

Use *return ici_int_ret(n);* to return an integer from an intrinsic fuction.

ici_interface_check

```
int ici_interface_check(unsigned long mver, unsigned long
bver, char const *name)
```

Check that the seperately compiled module that calls this function has been compiled against a compatible versions of the ICI core that is now trying to load it. An external module should call this like:

```
if (ici_interface_check(ICI_VER, ICI_BACK_COMPAT_VER,
"myname"))
    return NULL;
```

As soon as it can on load. ICI_VER and ICI_BACK_COMPAT_VER come from ici.h at the time that module was compiled. This functions compares the values passed from the external modules with the values the core was compiled with, and fails (usual conventions) if there is any incompatibility.

ici_interface_check

```
int ici_interface_check(unsigned long mver, unsigned long
bver, char const *name)
```

Check that the seperately compiled module that calls this function has been compiled against a compatible versions of the ICI core that is now trying to load it. An external module should call this like:

```
if (ici_interface_check(ICI_VER, ICI_BACK_COMPAT_VER,
"myname"))
    return NULL;
```

As soon as it can on load. ICI_VER and ICI_BACK_COMPAT_VER come from ici.h at the time that module was compiled. This functions compares the values passed from the external modules with the values the core was compiled with, and fails (usual conventions) if there is any incompatibility.

ici_method

```
int ici_method(ici_obj_t *inst, ici_str_t *mname, char
*types, ...)
```

Call the method *mname* of the object *inst* with simple argument marshalling.

See *ici_func()* for an explanation of *types*. Apart from calling a method, this function behaves in the same manner as *ici_func()*.

ici_module_new

```
ici_objwsup_t * ici_module_new(ici_cfunc_t *cf)
```

Create a new module struct and assign the given cfuncs into it (as in ici_assign_cfuncs()). Returns NULL on error, usual conventions. The returned struct has an incref the caller owns.

ici_module_new

```
ici_objwsup_t * ici_module_new(ici_cfunc_t *cf)
```

Create a new module struct and assign the given cfuncs into it (as in ici_assign_cfuncs()). Returns NULL on error, usual conventions. The returned struct has an incref the caller owns.

ici_nalloc

```
void * ici_nalloc(size_t z)
```

Allocate an object of the given *size*. Return NULL on failure, usual conventions. The resulting object must be freed with ici_nfree() and only ici_nfree(). Note that ici_nfree() also requires to know the size of the object being freed.

This function is preferable to ici_alloc(). It should be used if you can know the size of the allocation when the free happens so you can call ici_nfree(). If this isn't the case you will have to use ici_alloc().

See also: *ICIs allocation functions*, ici_talloc(), ici_alloc(), ici_nfree().

ici_nfree

```
void ici_nfree(void *p, size_t z)
```

Free an object allocated with ici_nalloc(). The *size* passed here must be exactly the same size passed to ici_nalloc() when the allocation was made. If you don't know the size, you should have called ici_alloc() in the first place.

See also: *ICIs allocation functions*, ici_nalloc().

ici_obj

```
struct ici_obj
{
    char        o_tcode;
    char        o_flags;
    char        o_nrefs;
    char        o_leafz;
}
```

This is the universal header of all objects. Each object includes this as a header. In the real structures associated with each object type the type specific stuff follows

o_tcode — The small integer type code that characterises this object. Standard core types have well known codes identified by the TC_* defines below. Other types are registered at run-time and are given the next available code.

This code can be used to index ici_types[] to discover a pointer to the type structure (see above).

o_flags — Some boolean flags. Well known flags that apply to all object occupy the lower 4 bits of this byte. The upper four bits are available for object specific use. See O_* below.

o_nrefs — A small integer count of the number of references to this object that are *not* otherwise visible to the garbage collector.

o_leafz — If (and only if) this object does not reference any other objects (i.e. its t_mark() function just sets the O_MARK flag), and its memory cost fits in this signed byte (< 127), then its size can be set here to accelerate the marking phase of the garbage collector. Else it must be zero.

ici_obj

```
struct ici_obj
{
    char        o_tcode;
    char        o_flags;
    char        o_nrefs;
    char        o_leafz;
}
```

This is the universal header of all objects. Each object includes this as a header. In the real structures associated with each object type the type specific stuff follows

o_tcode — The small integer type code that characterises this object. Standard core types have well known codes identified by the TC_* defines below. Other types are registered at run-time and are given the next available code.

This code can be used to index ici_types[] to discover a pointer to the type structure (see above).

o_flags	Some boolean flags. Well known flags that apply to all object occupy the lower 4 bits of this byte. The upper four bits are available for object specific use. See O_* below.
o_nrefs	A small integer count of the number of references to this object that are *not* otherwise visible to the garbage collector.
o_leafz	If (and only if) this object does not reference any other objects (i.e. its t_mark() function just sets the O_MARK flag), and its memory cost fits in this signed byte (< 127), then its size can be set here to accelerate the marking phase of the garbage collector. Else it must be zero.

ici_objname

```
char * ici_objname(char p[ICI_OBJNAMEZ], ici_obj_t *o)
```

Format a human readable version of the object in less than 30 chars. Returns *p*.

ici_register_type

```
int ici_register_type(ici_type_t *t)
```

Register a new *ici_type_t* structure and return a new small int type code to use in the header of objects of that type. The pointer *t* passed to this function is retained and assumed to remain valid indefinetly (it is normally a statically initialised structure).

Returns the new type code, or zero on error in which case ici_error has been set.

ici_register_type

```
int ici_register_type(ici_type_t *t)
```

Register a new *ici_type_t* structure and return a new small int type code to use in the header of objects of that type. The pointer *t* passed to this function is retained and assumed to remain valid indefinetly (it is normally a statically initialised structure).

Returns the new type code, or zero on error in which case ici_error has been set.

ici_ret_no_decref

```
int ici_ret_no_decref(ici_obj_t *o)
```

General way out of an intrinsic function returning the object o where the given object has no extra refernce count. Returns 0 indicating no error.

This is suitable for using as a return from an intrinsic function as say:

```
return ici_ret_no_decref(o);
```

If the object you are returning has an extra reference which must be decremented as part of the return, use ici_ret_with_decref() (above).

ici_ret_with_decref

```
int ici_ret_with_decref(ici_obj_t *o)
```

General way out of an intrinsic function returning the object *o*, but the given object has a reference count which must be decref'ed on the way out. Return 0 unless the given *o* is NULL, in which case it returns 1 with no other action.

This is suitable for using as a return from an intrinsic function as say:

```
return ici_ret_with_decref(objof(ici_int_new(2)));
```

(Although see ici_int_ret().) If the object you wish to return does not have an extra reference, use ici_ret_no_decref().

ici_str_alloc

```
ici_str_t * ici_str_alloc(int nchars)
```

Allocate a new string object (single allocation) large enough to hold nchars characters, and register it with the garbage collector. Note: This string is not yet an atom, but must become so as it is *not* mutable.

WARINING: This is *not* the normal way to make a string object. See ici_str_new().

ici_str_buf_new

```
ici_str_t * ici_str_buf_new(int n)
```

Return a new mutable string (i.e. one with a seperate growable allocation). The initially allocated space is n, but the length is 0 until it has been set by the caller.

The returned string has a reference count of 1 (which is caller is expected to decrement, eventually).

Returns NULL on error, usual conventions.

ici_str_get_nul_term

```
ici_str_t * ici_str_get_nul_term(char *p)
```

Make a new atomic immutable string from the given characters.

The returned string has a reference count of 0, unlike ici_str_new_nul_term() which is exactly the same in other respects.

Returns NULL on error, usual conventions.

ici_str_need_size

```
int ici_str_need_size(ici_str_t *s, int n)
```

Ensure that the given string has enough allocated memory to hold a string of n characters (and a guard *0* which this routine stores). Grows ths string as necessary. Returns 0 on success, 1 on error, usual conventions. Checks that the string is mutable and not atomic.

ici_str_need_size

```
int ici_str_need_size(ici_str_t *s, int n)
```

Ensure that the given string has enough allocated memory to hold a string of n characters (and a guard *0* which this routine stores). Grows ths string as necessary. Returns 0 on success, 1 on error, usual conventions. Checks that the string is mutable and not atomic.

ici_str_new

```
ici_str_t * ici_str_new(char *p, int nchars)
```

Make a new atomic immutable string from the given characters.

Note that the memory allocated to a string is always at least one byte larger than the listed size and the extra byte contains a *0*. For when a C string is needed.

The returned string has a reference count of 1 (which is caller is expected to decrement, eventually).

Returns NULL on error, usual conventions.

ici_str_new_nul_term

```
ici_str_t * ici_str_new_nul_term(char *p)
```

Make a new atomic immutable string from the given characters.

The returned string has a reference count of 1 (which is caller is expected to decrement, eventually).

Returns NULL on error, usual conventions.

ici_str_ret

```
int ici_str_ret(char *str)
```

Use *return ici_str_ret(n);* to return a string from an intrinsic fuction.

ici_str_ret

```
int ici_str_ret(char *str)
```

Use *return ici_str_ret(n);* to return a string from an intrinsic fuction.

ici_type

```
struct ici_type
{
```

```
    unsigned long (*t_mark)(ici_obj_t *);
    void        (*t_free)(ici_obj_t *);
    unsigned long (*t_hash)(ici_obj_t *);
    int         (*t_cmp)(ici_obj_t *, ici_obj_t *);
    ici_obj_t   *(*t_copy)(ici_obj_t *);
    int         (*t_assign)(ici_obj_t *, ici_obj_t *,
ici_obj_t *);
    ici_obj_t   *(*t_fetch)(ici_obj_t *, ici_obj_t *);
    char        *t_name;
    void        (*t_objname)(ici_obj_t *, char
[ICI_OBJNAMEZ]);
    int         (*t_call)(ici_obj_t *, ici_obj_t *);
    ici_str_t   *t_ici_name;
    int        (*t_assign_super)(ici_obj_t *, ici_obj_t *,
ici_obj_t *, ici_struct_t *);
    int         (*t_fetch_super)(ici_obj_t *, ici_obj_t *,
ici_obj_t **, ici_struct_t *);
    int         (*t_assign_base)(ici_obj_t *, ici_obj_t *,
ici_obj_t *);
    ici_obj_t   *(*t_fetch_base)(ici_obj_t *, ici_obj_t *);
    ici_obj_t   *(*t_fetch_method)(ici_obj_t *, ici_obj_t
*);
    void        *t_reserved2;   /* Must be zero. */
    void        *t_reserved3;   /* Must be zero. */
    void        *t_reserved4;   /* Must be zero. */
 }
```

Every object has a header. In the header the o_tcode (type code) field can be used to index the ici_types[] array to discover the obejct's type structure. This is the type structure. See detailed comments below.

t_mark(o)

Sets the O_MARK flag in o->o_flags of this object and all objects referenced by this one which don't already have O_MARK set. Returns the approximate memory cost of this and all other objects it sets the O_MARK of. Typically recurses on all referenced objects which don't already have O_MARK set (this recursion is a potential problem due to the uncontrolled stack depth it can create). This is used in the marking phase of garbage collection.

The macro ici_mark() calls the t_mark function of the object (based on object type) if the O_MARK flag of the object is clear, else it returns 0. This is the usual interface to an object's mark function.

The mark function implemantation of objects can assume the O_MARK flag of the object they are being invoked on is clear.

t_free(o)

Frees the object o and all associated data, but not other objects which are referenced from it. This is only called from garbage collection. Care should be taken to remember that errors can occur during object creation and that the free function might be asked to free a partially allocated object.

t_cmp(o1, o2)

Compare o1 and o2 and return 0 if they are the same, else non zero. This similarity is the basis for merging objects into single atomic objects and the implementation of the == operator.

Some objects are by nature both unique and intrinsically atomic (for example, objects which are one-to-one with some other allocated data which they alloc when the are created and free when they die). For these objects the existing function ici_cmp_unique() can be used as their implementation of this function.

It is very important in implementing this function not to miss any fields which may otherwise distinguish two obejcts. The cmp, hash and copy operations of an object are all related. It is useful to check that they all regard the same data fields as significant in performing their operation.

t_copy(o)

Returns a copy of the given object. This is the basis for the implementation of the copy() function. On failure, NULL is returned and error is set. The returned object has been ici_incref'ed. The returned object should cmp() as being equal, but be a distinct object for objects that are not intrinsically atomic.

Intrinsically atomic objects may use the existing function ici_copy_simple() as their implemenation of this function.

t_hash(o)

Return an unsigned long hash which is sensitive to the value of the object. Two objects which cmp() equal should return the same hash.

The returned hash is used in a hash table shared by objects of all types. So, somewhat problematically, it is desireable to generate hashes which have good spread and seperation across all objects.

Some objects are by nature both unique and intrinsically atomic (for example, objects which are one-to-one with some other allocated data which they alloc when the are created and free when they die). For these objects the existing function hash_unique() can be used as their implementation of this function.

t_assign(o, k, v)

Assign to key k of the object o the value v. Return 1 on error, else 0.

The existing function ici_assign_fail() may be used both as the implementation of this function for object types which do not support any assignment, and as a simple method of generating an error for particular assignments which break some rule of the object.

Not that it is not necessarilly wrong for an intrinsically atomic object to support some form of assignment. Only for the modified field to be significant in a cmp() operation. Objects which are intrinsically unique and atomic often support assignments.

t_fetch(o, k)

Fetch the value of key k of the object o. Return NULL on error.

Note that the returned object does not have any extra reference count; however, in some circumstances it may not have any garbage collector visible references to it. That is, it may be vunerable to a garbage collection if it is not either incref()ed or hooked into a referenced object immediately. Callers are responsible for taking care.

The existing function ici_fetch_fail() may be used both as the implementation of this function for object types which do not support any assignment, and as a simple method of generating an error for particular fetches which break some rule of the object.

t_name — The name of this type. Use for the implementation of typeof() and in error messages. But apart from that, type names have no fundamental importance in the langauge and need not even be unique.

t_objname(o, p) — Place a short (30 chars or less) human readable representation of the object in the given buffer. This is not intended as a basis for re-parsing or serialisation. It is just for diagnostics and debug. An implementation of t_objname() must not allocate memory or otherwise allow the garbage collector to run. It is often used to generate formatted failure messages after an error has occured, but before clean-up has completed.

t_call(o, s) — Call the object o. If s is non-NULL this is a method call and s is the subject object of the call. Return 1 on error, else 0.

t_ici_name — A ici_str_t copy of the name. This is just a cached version so that typeof() doesn't keep re-computing the string.

t_fetch_method — An optional alternative to the basic t_fetch() that will be called (if supplied) when doing a fetch for the purpose of forming a method. This is really only a hack to support COM under Windows. COM allows remote objects to have properties, like object.property, and methods, like object:method(). But without this special hack, we can't tell if a fetch operation is supposed to perform the COM get/set property operation, or return a callable object for a future method call. Most objects will leave this NULL.

ici_typecheck

```
int ici_typecheck(char *types, ...)
```

Marshall function argument in a call from ICI to C. The argspec is a character string. Each character corresponds to an actual argument to the ICI function which will (may) be assigned through the corresponding pointer taken from the subsequent arguments. Any detected type mismatches result in a non-zero return. If all types match, all assignments will be made and zero will be returned.

The argspec key letters and their meaning are:

o — Any ICI object is required in the actuals, the corresponding pointer must be a pointer to an *(ici_obj_t *)*; which will be set to the actual argument.

h	An ICI handle object, that has the name given by the corresponding argument, is required. The next argument is a pointer to store the *(handle_t *)* through.
p	An ICI ptr object is required in the actuals, then as for o.
d	An ICI struct object is required in the actuals, then as for o.
a	An ICI array object is required in the actuals, then as for o.
u	An ICI file object is required in the actuals, then as for o.
r	An ICI regexp object is required in the actuals, then as for o.
m	An ICI mem object is required in the actuals, then as for o.
i	An ICI int object is required in the actuals, the value of this int will be stored through the corresponding pointer which must be a (long *).
f	An ICI float object is required in the actuals, the value of this float will be stored through the corresponding pointer which must be a (double *).
n	An ICI float or int object is required in the actuals, the value of this float or int will be stored through the corresponding pointer which must be a (double *).
s	An ICI string object is required in the actuals, the corresponding pointer must be a (char **). A pointer to the raw characters of the string will be stored through this (this will be *0* terminated by virtue of all ICI strings having a gratuitous *0* just past their real end).
-	The acutal parameter at this position is skipped, but it must be present.
*	All remaining actual parametes are ignored (even if there aren't any).

The capitalisation of any of the alphabetic key letters above changes their meaning. The acutal must be an ICI ptr type. The value this pointer points to is taken to be the value which the above descriptions concern themselves with (i.e. in place of the raw actual parameter).

There must be exactly as many actual arguments as key letters unless the last key letter is a *.

Error returns have the usual ICI error conventions.

Building ICI on various platforms

Windows

Coming soon.

Some tips for debugging extension modules in Visual C: In order to make sure that the ICI executable loads the debug version you have built (rather than an installed version of ICI), do this: For *Program arguments* in the Settings/Debug/General tab, use:

```
-e "rpush(path, \"Debug\");" -f test.ici
```

for the Debug build, and:

```
-e "rpush(path, \"Release\");" -f test.ici
```

for the Release build.

UNIX-like systems

Coming soon.

How it works

These are notes for a new chapter. Cover:

- Implementation of parser/compiler and execution engine using the common data structures.
- Operation of the execution engine.
- Logic behind objects semantics - single pointer, no special cases.
- Garbage collector.
- Lookup-lookaside.