ICS

Development System
Version 2.1

Revision 1.2 / 5.3.2004

Contents

Introduction

The **i**nfoGrips **C**onversion **S**ystem (ICS) is a flexible environment for the fast development of GIS translators. The following translators have been developed with ICS:

- INTERLIS => Microstation
- Microstation => INTERLIS
- ADALIN => INTELIS
- ARGIS => INTERLIS
- INTERLIS => ARGIS
- INFOCAM => INTERLIS
- INTERLIS => INTERLIS
- etc.[1]

ICS translators are best known for their flexibility. This flexibility is mainly achieved by the built in scripting language iG/Script [1]. With iG/Script an end user can configure ICS to his or her needs. The configuration and even the programming of ICS with iG/Script is highly efficient as long as all required input and output modules for a given translator exist. If on the other hand one required module misses, or the user needs some special functionality, the need for the development of new ICS modules becomes obvious.

Intended Audience

This documentation addresses developers who need to extend the basic functionality of ICS by own modules. ICS modules are normally written in the programming language C or C++. The reader of this document should therefore be familiar with C or C++. Also a good knowledge of the scripting language iG/Script is necessary to fully understand this documentation. Readers not familiar with iG/Script or C/C++ should consult the related documentations [1][2].

Organisation of this Documentation

This manual is structured as follows:

- Part 1 gives an introduction to ICS and it's concepts.
- Part 2 explains all the necessary steps to develop a new ICS translators.
- Part 3 contains a comprehensive list of all ICS kernel functions available to a developer
- In the appendix you will find printed listings of the examples presented in this documentation and a list of related literature.

Conventions

The following conventions are used in this manual:

italics	filenames
bold	new terms, names of C-functions or ICS methods
`courier`	program text or operating system commands

[1] ICS is fully independent of the modeling language INTERLIS, but ICS has the required flexibility to master all the features of INTERLIS. That's why ICS has been primarily used to implement INTERLIS translators.

Other Documentations

This documentation is supplemented by the following documentations:

∇ iG/Script Benutzer- und Referenzhandbuch (in german) [1]

∇ ICS Benutzerhandbuch (in german) [3]

Supported Platforms

The ICS development system is available for the following operating systems:

∇ Windows NT (3.51 and 4.0)

∇ Windows 95

Other operating systems may be supported in the future.

Part I: Fundamentals

1. ICS Concepts

1.1 Introduction

GIS translators are written today as C, FORTRAN or BASIC programs or are coded with scripting languages like awk and Pearl. The usage of 3rd generation language like C normally leads to very specific solutions, which are adaptable only in a limited range. Modifications of the GIS data model are often followed by modification of the translator source code and recompilation. The high cost of this approach is well known. Scripting languages are easier to adapt to new situations. Unfortunately scripting languages do not support GIS data types like point, line and area or special operations like topology generation and database access. Normally there is also no way to extend scripting languages by user programmed **external modules**[2,3]. The implication is that translator developers have to live with the limitations of the scripting language. The scripting approach is therefore primarily used in quick and dirty ad hoc solutions.

With the development of ICS (infoGrips Conversion System) we wanted to combine the advantages of interpreted and compiled languages to support the fast and flexible development of GIS translators. The ICS kernel contains an interpreter for the scripting language **iG/Script** which is specially optimized for GIS translator applications (i.e. includes GIS data types, database access, topology generation, etc.). Still we think that the developer needs the possibility to extend iG/Script by own external modules. That's why we have included in ICS the possibility to load user programmed external modules at runtime. So even if iG/Script doesn't solve a particular problem directly, the developer has the possibility to code the solution by an external module.

1.2 ICS Architecture

After the general introduction we will have now a closer look on the ICS architecture and the way how ICS works. The main idea behind ICS is the observation that every GIS translator can be split in three modules input, output and main.

- ∇ The **input module** reads input objects from the input source (file, database, or GIS system).
- ∇ The **output module** writes objects to the destination (file, database or GIS system).

[2] A notable exception from this rule is TCL (Tool Command Language)
[3] An external module is a module written in an other language which implements some new functionality for the scripting language.

∇ The **main module** controls the data flow. It reads input objects from the source, manipulates them, and sends the objects after manipulation to the destination for final storage.

This is observation is quite general, and is implemented of course more or less elegant in every GIS translator system. In ICS we have realized this concept as follows:

∇ Input modules are implemented in ICS by external modules. Under Windows NT and Windows 95 external modules are implemented as dynamic load libraries (DLL's). This means, that external modules can be loaded dynamically at runtime.

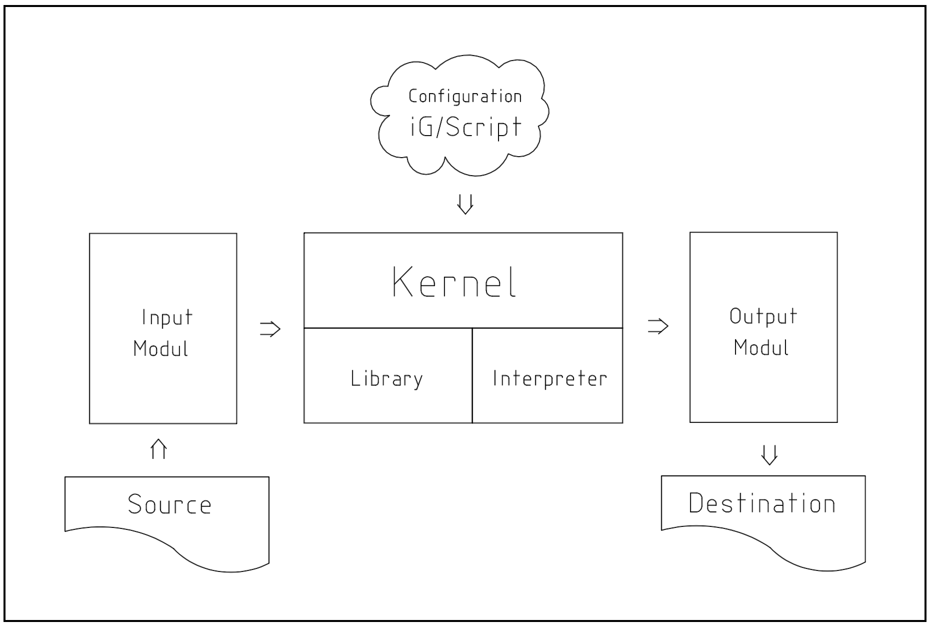
∇ Output modules are also implemented in ICS by external modules. In fact there's not much difference between an output and an input module, besides the fact that an input module normally reads from the input source and an output module mainly writes to the output destination.

∇ The main module is implemented by the **ICS kernel**. The ICS kernel consists of a large library of predefined functions (type conversions, geometry, topology, file access, database access etc.). The kernel also directly supports all needed GIS data types (point, line, area). The communication between kernel and external modules is done by an **common object model**.

∇ The kernel also contains an interpreter for the scripting language iG/Script which controls the kernel at runtime. The user can thus configure the ICS system entirely by iG/Script. All required manipulations of objects (calculations, mapping of values, access or storage of objects etc.) can be done within iG/Script.

∇ The predefined library functions can be used either by an external module or through the scripting language iG/Script.

The following figure shows the general architecture of the ICS system:



Remarks:

∇ In the case of changed translator requirements (i.e. changed datamodel), only the scripts have to be adapted, which can be done quickly by the user (no developer assistance is necessary). The kernel, input and output modules are normally not affected by modifications.

∇ An other advantage of this architecture is the following. Assume that two input modules I1 and I2 and two output modules O1 and O2 have been realized, than all possible combinations of translators (I1 => O1, I1 => O2, I2 => O1, I2 => O2) can be realized by different configuration scripts. No additional coding in C/C++ is necessary.

1.3 Common Object Model

A key component of the ICS architecture is the common object model. Everything inside the ICS kernel and iG/Script is treated as an object. The ICS kernel and the external modules communicate through globally named objects.

An ICS object is a flexible data structure, that encompasses all needed GIS object types. The following properties hold for ICS objects:

- ∇ ICS Objects consist of components. A component has a name and a value within the object.
- ∇ The value may have one of the following **base types**: Integer, Real, Boolean, String, Point, Line, Area and Map (Object).
- ∇ The value of a component may be also a complete object (complex object type).
- ∇ In general an object can be hierarchical. The number of nesting levels is not limited.
- ∇ The number of components per object and the component types are not limited (i.e. an object can have several geometric components).
- ∇ Objects can have global names (i.e. IN or OUT). All globally named objects can be accessed and manipulated through iG/Script.

Next follows an example of a global object IN delivered by the INTERLIS input module ILIN:

object IN	
MODEL	'Grunddatensatz'
TOPIC	'Fixpunkte'
TABLE	'LFP'
OBJID	'101'
geometry	675899.226/245270.946
number	'1091111.2'
numberpos	675890.222/235271.344
etc.	…

Our experience shows that this object model is sufficient to describe all current GIS systems.

2. An introductory Example

The explanations so far may still seem a little bit abstract. So let us look at a short example.

Example:

We like to develop a translator from INTERLIS to DXF. Point objects read from the input file should be written as DXF blocks to the output.

Solution:

We have to implement the input module `ILIN` (INTERLIS input) and the output module `DXFOUT` (DXF output) as external modules (written in C). For this example we will assume that they already exist. The `ILIN` module implements the iG/Script methods[4] `ILIN.OPEN`, `ILIN.CLOSE` and `ILIN.READ_OBJECT`. The `DXFOUT` module implements

[4] iG/Script methods are introduced later. For now it is sufficient to assume that there is some way to implement methods in an external module, that can be accessed by iG/Script.

the methods `DXFOUT.OPEN`, `DXFOUT.CLOSE` and `DXFOUT.WRITE_OBJECT`. The iG/Script program which controls the ICS kernel can now be written as follows (for simplicity we have omitted parameter passing and error handling):

```
|LOAD ilin,dxfout               ! dynamic loading of ILIN and DXFOUT

'input.itf' ILIN.OPEN           ! open the input file
'output.dxf' DXFOUT.OPEN        ! create an output file

WHILE ILIN.READ_OBJECT DO       ! read the next object from the input source
      'BLOCK' => OUT.TYPE       ! create an output object OUT
      IN.Lage => OUT.GEOM       !
      'Pkt' => OUT.BLOCK        !
      DXFOUT.WRITE_OBJECT       ! write that object to the output stream
END_WHILE

ILIN.CLOSE                      ! close the input
DXFOUT.CLOSE                    ! close the output
```

This example shows very well the general structure of an iG/Script configuration program. First input source and output destination are opened. Second a main loop reads objects from the input and sends them to the output. Inside the loop the necessary manipulations of the objects are done with iG/Script. Finally the input source and the output destinations are closed. The user of iG/Script does not need to know all the details of the input format (INTERLIS) and the output format (DXF). All the format specific details, like invocation of the INTERLIS compiler, the generation of the DXF header and the correct formatting and ordering of the objects on the output stream are handled by the input and output modules respectively.

Remark: The end user of an ICS translator, does not need to know how to program a main loop in iG/Script. All our translators are delivered with a base configuration in iG/Script and user manuals which describe how to adapt these base configurations to special needs.

Part II: Application Development

3. Application Development with ICS

3.1 Introduction

An ICS translator application consist of an input module, an output module and an iG/Script configuration. The necessary implementation steps of input and output modules are described in this chapter. The iG/Script language is described in [1]. A complete example is given in chapter 4.

3.2 Development Steps

The development process of an ICS translator application consists normally of the following steps:

1. **Analysis**. The translator application is analyzed. The input and output modules are identified and the necessary methods are defined. The required input and output objects are also defined in this step.
2. **Implementation of the input module**. The input methods and objects are implemented in C by an external module.
3. **Implementation of the output module.** The output methods and objects are implemented in C by an external module.
5. **iG/Script configuration**. Next all parts of the ICS application are integrated by an iG/Script configuration.
6. **Application testing**. You should test your application step by step. First test the input module and output modules separately. Second test the whole application.
7. **Documentation**. Document your application.

3.3 Implementation of External Modules

3.3.1 Required C Functions

The following C functions have to be implemented for all external modules (don't forget to include these functions in the export list of your DLL's):

Function:
```
ICS_VOID DLLEXPORT <MODULE>_INIT(
      ICS_CHAR *mainlib
);
```
Description: Initializes the module. This function is called right after the kernel has loaded the external module. In this function the module first attaches itself to the kernel library (mainlib). Second all iG/Script methods implemented by the module are registered.

Funktion:
```
ICS_VOID DLLEXPORT <MODULE>_TERMINATE(
);
```
Description: Terminates the module. This function is called by the kernel when the external module is not needed anymore. It is a good idea to release all allocated resources (memory, files etc.) in this step.

3.3.2 Global Maps

Usually external modules should support the global map `<MODULE>_PARAM`. In this map the user of the module can pass parameters to the external module. The module should support at least the parameter `DEBUG`.

3.3.3 Generating external Modules

External modules have to be compiled as 32bit DLL's. The `<MODULE>_INIT()` and `<MODULE>_TERMINATE()` functions have to be included in the export list of the DLL. If you forget to export the function, the ICS kernel is unable to load the external module and a runtime error is displayed.

3.4 Implementation of Input Modules

3.4.1 Input Methods

The following iG/Script methods have to be implemented by input modules:

Method: **`<MODULE>.OPEN`** [s filename] []
Description: Opens the input source. This method is called by an iG/Script program to start reading from an input source.

Method: **`<MODULE>.READ_OBJECT`** [] [b ok]
Description: Reads the next object from the opened input source. The object has to be returned in the predefined global Map `IN` (the method should first clear the map `IN` before it fills in new components). If the operation was successful, the input module should return `ICS_TRUE` on the ICS stack.

Method: **`<MODULE>.CLOSE`** [] []
Description: Closes the input source.

3.4.2 Map IN

Every input module has to process the global map IN. You can fill object components in the map IN as follows:

1) Retrieve a `MAP_MAP` handle to the map `IN` by `MAP_GET_MAP_BY_NAME_C()`. Store this handle in a global variable because you will need the map `IN` a lot of times.
2) Insert your components with `MAP_INSERT_VALUE_C()` (for string components) or `MAP_INSERT_OBJECT_C()` (for all other types).

3.5 Implementation of Output Modules

3.5.1 Output Methods

The following iG/Script methods have to be implemented by output modules:

Method:	**`<MODULE>.OPEN`** [s filename] []
Description:	Opens the output destination. This method is called by iG/Script programs to start writing to an output destination.
Method:	**`<MODULE>.WRITE_OBJECT`** [] []
Description:	Writes an output object to the destination. The output object is passed to the method by the global map `OUT`.
Method:	**`<MODULE>.CLOSE`** [] []
Description:	Closes the output destination.

3.5.2 Map OUT

Output objects are passed to the output module by the global map `OUT`. You can access output object components as follows:

1) Retrieve a `MAP_MAP` handle to the map `OUT` by `MAP_GET_MAP_BY_NAME_C()`. Store this handle in a global variable because you will need the map `OUT` a lot of times.
2) Retrieve object components by `MAP_MAP_OBJECT_C()` or by `MAP_RESET()` followed by `MAP_READ_OBJECT_C()`.

4. A complete Example

4.1 Introduction

In chapter 4 we introduced application development with ICS. In this chapter we will present a comprehensive example. To keep the example as simple as possible, we will concentrate ourselves on a simple but useful example. The problem is defined as follows:

> Develop an ICS translator application for copying text files.

4.2 Analysis

Obviously to copy text files we have to read from an input text file and write an output text file. Therefore we implement an input module `CIN` and an output module `COUT`. The objects of our application are text lines. Every time the configuration script calls `CIN.READ_OBJECT`, the `CIN` module returns in `IN.TEXT` a new text line. In `COUT` we implement the method `COUT.WRITE_OBJECT` which expects a text line in `OUT.TEXT`.

List of all input module methods and objects to implement:

Methods:	`CIN.OPEN,CIN.CLOSE,CIN.READ_OBJECT`
Objects:	Map `IN` with component `TEXT`

List of all output module methods and objects to implement:

Methods:	`COUT.OPEN,COUT.CLOSE,COUT.WRITE_OBJECT`
Objects:	Map `OUT` with component `TEXT`

4.3 Coding of the Input Module

The complete C listing of module `CIN` is contained in appendix A2.

4.4 Coding of the Ouput Module

The complete C listing of module `COUT` is contained in appendix A3.

4.5 Testing

For testing we write two short iG/Script programs. *test1.cfg* tests the output module and *test2.cfg* tests the input module. The listing of the scripts is contained in appendix A4.1 and A4.2.

If you have installed the ICS Development System (see also Chapter 5.1), you can run the test scripts with ICS for Windows (subdirectory icsdev).

4.6 iG/Script Configuration

Finally we write the script *copy.cfg* (see appendix A5). The script can be executed by ICS for Windows.

Part III: ICS Library Reference

5. Introduction to the ICS Library

This chapter gives a general introduction to the ICS library and it's usage. All ICS data types and the ICS object structure are documented. A comprehensive list of all available ICS library function groups can be found in chapter 6.

5.1 Installation

The ICS Developmentsystem is installed as follows:

∇ Install the INTERLIS Tools [3].

  Ther ICS Developmentsystem is included in the product INTERLIS Tools.
  The expression `ILTOOLS_DIR` will be used to indicate the root-installation-directory of the product INTERLIS Tools.

After you have successfully installed the product, you have to create a new development project with your C/C++ compiler. Besides your own files you have to include at least the file *icslib.c* into your project (`ILTOOLS_DIR\`*icsdev\icslib.c*). You should also include the `ILTOOLS_DIR\`*icsdev\h* path as an additional search path for header files (**.h*). Finally you have to define the preprocessor variable `_ICS_NT` (`-D_ICS_NT`) for the project. The `_ICS_NT` preprocessor variable indicates to the ICS library that the target operating system is Windows NT or Windows 95.

You can create your external modules with any C/C++ compiler that is capable of generating Windows 32bit DLL's with `__stdcall` calling convention. We recommend Microsoft Visual C/C++ Version 2.0 or Watcom C/C++ Version 10.5 (these compilers have been tested with ICS). After compilation you have to copy the DLL's to a directory where Windows NT (or Windows 95) searches for DLL's (i.e. `ILTOOLS_DIR\`*bin*). If you forget this step the ICS kernel will not find your external modules and you will get a runtime error.

5.2 ICS Data Types

5.2.1 Basic Data Types

The ICS kernel is coded in an operating system and compiler independent way. The ICS kernel uses therefore only portable data types and not C data types (int, float, etc.). The following basic types are used by ICS:

`ICS_CHAR`	unsigned 8 bit ASCII character
`ICS_BOOLEAN`	Boolean type, variables of this type can only have the predefined values `ICS_TRUE` or `ICS_FALSE`
`ICS_INT`	16 bit Integer
`ICS_INT16`	16 bit Integer
`ICS_INT32`	32 bit Integer
`ICS_CARD16`	unsigned 16 bit Integer
`ICS_CARD32`	unsigned 32 bit Integer
`ICS_REAL64`	real value, 64 bit

All library functions need arguments of those types. If you don't want to use ICS types in your external modules, you have to cast your arguments while calling an ICS library function.

Example:

function:	`MESSAGE_DISPLAY_C(ICS_CHAR *, ...)`
proper call:	`MESSAGE_DISPLAY_C((ICS_CHAR *)"hello, World");`

5.2.2 Data Type STRNG_STRING

The kernel stores all string values as special data type STRNG_STRING (= kernel string). Such a kernel string is very different from a C string. Kernel strings can dynamically grow an shrink as needed, whereas C strings are statically allocated by the C compiler. Kernel strings are of interest to the ICS library user, because some ICS library functions need kernel strings as arguments or give them back as results. Of course it is possible to convert kernel strings to C strings and back by library functions. The ICS library offers also some functions to manipulate kernel strings directly (see string function group). The STRNG_STRING data type is defined in C as follows:

```
typedef struct STRNG_STRING {
      ICS_CARD16 AL;
      ICS_CARD16 LGTH;
      ICS_CHAR *ST;
} STRNG_STRING;
```

A variable of type STRNG_STRING has a string body ST and an allocated length LGTH. The pointer ST points to an allocated block of memory where the characters of the string are stored. AL holds the allocated size of this block. LGTH stores the actual size of the string. LGTH is equal or less than AL. This information is only given to better understand the kernel string type. The application developer should only manipulate kernel strings by library functions and never directly.

From the application developers point of view the most important thing with kernel strings is that kernel string always have to be created first with STRNG_CREATE(). After you don't need the string anymore you should delete it again by STRNG_REMOVE()(allocated memory is released again). If you access a kernel string that has not been created, unexpected things may happen (i.e. bad memory access).

5.2.3 Data Type GEOM_GEOMETRY

An other special data type supported by the kernel is GEOM_GEOMETRY (= geometry). The geometry data type is a generic data type to store point, line and area values. The geometry data type is built up hierarchical. The point is the basic structure. A point has always 3D coordinates. A line consists of points. The connections between points can be either linear or circular (the arc connection is defined by an intermediate point). Several lines can form a closed boundary (= RAND). An area finally consist of an outer boundary and 0-N inner island boundaries. The ICS library offers a comprehensive set of functions to manipulate the geometry data type.

5.2.4 Data Type MAP_MAP

The map data type is the basic type to build structured or hierarchical objects. A map consists of 0-N name/value pairs (components). A map can be compared with the C `struct` type. It may also be seen as mapping table, or a structured object. See [1] for a detailed discussion of the map data type. The ICS library provides a comprehensive set of functions to manipulate the map data type.

5.2.5 Data Type LIST_LIST

Sometimes it is desirable to organize Objects in lists. The list data type provides the necessary functionality. Note: Manipulation functions of the list data type are part of the `BTYP_*` function group.

5.3 ICS Objects

Finally we have to introduce ICS objects. As mentioned before, everything inside the kernel is treated as an object. Objects can be seen as containers for kernel type values (basic data type, string, geometry, map) with some additional administrative information. The administrative information includes the following:

- ∇ A type identifier (TID). The TID indicates what kind of data type the object contains and the kernel operations allowed on that object.
- ∇ An undefined indicator. If the undefined indicator is set for an object, the object value is undefined. Note that the kernel knows the type of an object by the TID even if the value is undefined. The undefined value therefore has a type in ICS. You can test the undefined status by the built in function `OBJ_object_value_is_defined()`.
- ∇ A reference count. An object may be referenced several times by the kernel or external modules. The kernel keeps track of all references to an object. If an object is not referenced anymore the kernel automatically deletes the object (semi automatic garbage collection).

Because C does not offer an object concept, ICS objects are implemented as a special data structure. The developer should manipulate this data structure only by ICS library functions and never directly.

The developer of external modules is confronted with objects in the following cases:

- ∇ The used library function is a generic function that operates on all kernel types (i.e. `OBJ_copy_object()`).
- ∇ The result type of a library function is not known in advance (i.e. `MAP_MAP_OBJECT()`, `ICSCPU_POP_OBJ()`).
- ∇ The developer want's to implement a generic iG/Script method, which operates on several kernel types (i.e. a calculation method that accepts integer and real values).

As with kernel strings it is extremely important to initialize objects first with `OBJ_init_objectvar()` and to release the object after it's not used anymore by `OBJ_remove_object()`. Because objects are normally used only inside the kernel, it is necessary to have operations to pack and unpack ordinary data types in objects. This pack and unpack operations are implemented in the `BTYP_*` function group.

6. ICS Library Function Groups

6.1 String Function Group (#include <*strng.h*>, <*strng2.h*>)

Function:

```
ICS_VOID STRNG_CREATE(
      STRNG_STRING *S
)
```

Description: Creates a new string. You should always create a string before you use it.

Function:

```
ICS_VOID STRNG_REMOVE(
      STRNG_STRING *S
)
```

Description: Deletes a string. If you don’t need a string anymore, clean it up with `STRNG_REMOVE()`.

Function:

```
ICS_VOID STRNG_COPY(
      STRNG_STRING S1,
      ICS_CARD16 LENGTH,
      STRNG_STRING *S2
)
```

Description: Makes a copy of string `S1` in string `S2`. In `LENGTH` you can specify how many characters of `S1` you wish to copy. If `LENGH` is `0`, all characters of `S1` are copied to `S2`.

Function:

```
ICS_VOID STRNG_APP(
      STRNG_STRING *S1
      STRNG_STRING *S2
)
```

Description: Concatenates two strings. S1 is appended to the end of S2.

Function:

```
ICS_VOID STRNG_FILL_C(
      ICS_CHAR * FILL,
      STRNG_STRING *S
)
```

Description: Fills a C string in a kernel string `S`. The old contents of the the kernel string is overwritten. You can use this function to convert C strings to kernel strings.

Example: `STRNG_FILL_C((ICS_CHAR *)"hello, World",&S)`

Function:

```
ICS_BOOLEAN STRNG2_MSTRING_TO_CSTRING(
      STRNG_STRING *KS,
      ICS_CHAR *CS
)
```

Description: Converts a kernel string `KS` to a C string `CS`.

Function:

```
ICS_BOOLEAN STRNG2_MATCHA_C(
      STRNG_STRING *S,
      ICS_CHAR *MATCH
)
```

Description: Compares a kernel string with a C string. If the strings are identically the result of the functions is `ICS_TRUE`, else `ICS_FALSE`.

6.2 Geometry Function Group (#include <geom.h>)

Function:

```
ICS_VOID GEOM_INITIALIZE(
      GEOM_GEOMETRY *G
);
```

Description: Initializes a variable of type geometry.

Function:

```
ICS_VOID GEOM_CREATE_POINT(
      GEOM_GEOMETRY *P,
      ICS_REAL64 X,
      ICS_REAL64 Y,
      ICS_REAL64 Z
);
```

Description: Creates a point with `X/Y/Z` coordinates.

Function:

```
ICS_VOID GEOM_CREATE_LINE(
      GEOM_GEOMETRY *L,
      GEOM_GEOMETRY P1,
      ICS_REAL64 ARCRAD,
      ICS_REAL64 PX,
      ICS_REAL64 PY,
      ICS_REAL64 PZ,
      GEOM_GEOMETRY P2
);
```

Description: Creates a line by two points `P1` and `P2`. If `ARCRAD` is != 0.0 the point defined by `PX/PY/PZ` is on the periphery of the arc defined by `P1`, `PX/PY/PZ` and `P2`. `ARCRAD` is the absolute value of the circle radius.

Function:

```
ICS_VOID GEOM_APPEND_LINE_POINT(
      GEOM_GEOMETRY *L,
      ICS_REAL64 ARCRAD,
      ICS_REAL64 PX,
      ICS_REAL64 PY,
      ICS_REAL64 PZ,
      GEOM_GEOMETRY P
);
```

Description: Appends a point at the end of an existing line. If `ARCRAD` is != 0.0 an arc connection is appended with `PX/PY/PZ` as a periphery point.

Function:

```
ICS_VOID GEOM_CREATE_RAND(
      GEOM_GEOMETRY *R,
      GEOM_GEOMETRY L
);
```

Description: Creates a boundary (RAND) from an existing line. If the boundary consists only of one line the starting and end points of the line have to be identical.

Function:
```
ICS_VOID GEOM_APPEND_RAND_LINE(
      GEOM_GEOMETRY *R,
      GEOM_GEOMETRY L
);
```
Description: Appends a line to an existing boundary. If a boundary consists of more than one line, the lines have to form a closed ring, i.e. the end point of the first line is the starting point of the second line etc. and the end point of the last line is the starting point of the first line.

Function:
```
ICS_VOID GEOM_CREATE_AREA(
      GEOM_GEOMETRY *A,
      GEOM_GEOMETRY R
);
```
Description: Creates an area. `R` is the outer boundary of the area `A`.

Function:
```
ICS_VOID GEOM_APPEND_AREA_RAND(
      GEOM_GEOMETRY *A,
      GEOM_GEOMETRY R
);
```
Description: Inserts an island into an existing area.

Function:
```
ICS_VOID GEOM_READ_TYPE(
      GEOM_GEOMETRY G,
      GEOM_GEOMTYPE *T
);
```
Description: Delivers the type `T` (`GEOM_POINT`, `GEOM_LINE`, `GEOM_RAND` or `GEOM_AREA`) for a given geometry `G`.

Example:
```
GEOM_READ_TYPE(G,&T);
if (T == GEOM_POINT) {
      MESSAGE_DISPLAY_C("geometry is a point\n");
}
else {
      MESSAGE_DISPLAY_C("geometry is not a point\n");
}
```

Function:
```
ICS_VOID GEOM_READ_POINT(
      GEOM_GEOMETRY P,
      ICS_REAL64 *X,
      ICS_REAL64 *Y,
      ICS_REAL64 *Z
);
```
Beschreibung: Reads the coordinates `X/Y/Z` of a point.

Function:
```
ICS_VOID GEOM_RESET_READ(
      GEOM_GEOMETRY *G
);
```
Description: Prepares reading operations of subobjects for lines, boundaries and areas. After `GEOM_RESET_READ()` the reading operations return the first subobject.

Function:
```
ICS_VOID GEOM_READ_LINE_POINT(
      GEOM_GEOMETRY *L,
      ICS_REAL64 *ARCRAD,
      ICS_REAL64 *PX,
      ICS_REAL64 *PY,
      ICS_REAL64 *PZ,
      GEOM_GEOMETRY *P,
      ICS_BOOLEAN *OK
);
```

Description: Reads the next point `P` of a line. If `ARCRAD` is != 0.0 the connection from the last point to the actual point is an arc. The point `PX/PY/PZ` is on the periphery of the arc. If `OK` is `ICS_FALSE` there is no next point in the line. If you want to read the line again you have to call first `GEOM_RESET()` on that line.

Function:
```
ICS_VOID GEOM_READ_RAND_LINE(
      GEOM_GEOMETRY *R,
      GEOM_GEOMETRY *L,
      ICS_BOOLEAN *OK
);
```
Description: Reads the next line of a boundary. If `OK` is `ICS_FALSE` there is no next line in the boundary.

Function:
```
ICS_VOID GEOM_READ_AREA_RAND(
      GEOM_GEOMETRY *A,
      GEOM_GEOMETRY *R,
      ICS_BOOLEAN *OK
);
```
Description: Reads the next boundary of an area. The outer boundary is delivered first, the island boundaries are given back next. If `OK` is `ICS_FALSE` there is no next boundary in the area.

Function:
```
ICS_VOID GEOM_COPY(
      GEOM_GEOMETRY G1,
      GEOM_GEOMETRY *G2
);
```
Description: Makes a full copy of `G1` in `G2`.

Function:
```
ICS_VOID GEOM_REMOVE(
      GEOM_GEOMETRY *G
);
```
Description: Releases the memory occupied by geometry `G`.

6.3 Map Function Group (#include <map.h>)

Function:
```
ICS_VOID MAP_CREATE_C(
      MAP_MAP * M,
      ICS_CHAR *NAME
)
```
Description: Creates a global map `M` with name `NAME`. The new map is also accessible in iG/Script through it’s name `NAME`.

Example: `MAP_CREATE_C(&M,(ICS_CHAR *)"TEST");`

Function:
```
ICS_VOID MAP_CREATE_NONAME(
      MAP_MAP * M
)
```
Description: Creates a local map `M`. The map can not be accessed directly by iG/Script.

Example: `MAP_CREATE_NONAME(&M);`

Function:

```
ICS_VOID MAP_GET_MAP_BY_NAME_C(
      MAP_MAP *M,
      ICS_CHAR *NAME,
      ICS_BOOLEAN *OK
)
```

Description: Searches a global map `M` by it's name `NAME`. If the map exists, `OK` is `ICS_TRUE` and `M` points to the map found.

Example: `MAP_GET_MAP_BY_NAME_C(&M,(ICS_CHAR *)"TEST",&OK);`

Function:

```
ICS_VOID MAP_INSERT_VALUE_C(
      MAP_MAP * M,
      ICS_CHAR *NAME,
      ICS_CHAR *VALUE
)
```

Description: Inserts a new name/value pair in an existing map. The map may be local or global.

Example: `MAP_INSERT_VALUE_C(&M,(ICS_CHAR *)"T",(ICS_CHAR *)"TEST");`

Function:

```
ICS_VOID MAP_INSERT_OBJECT_C(
      MAP_MAP * M,
      ICS_CHAR *NAME,
      OBJ_Object VALUE
)
```

Description: Inserts a new name, value pair in an existing map. The map may be local or global. Use this function if you want to insert non string values in maps. `MAP_INSERT_OBJECT_C()` inserts a full copy of the `VALUE` object.

Example: `MAP_INSERT_OBJECT_C(&M,(ICS_CHAR *)"T",OBJ);`

Function:

```
ICS_VOID MAP_INSERT_OBJECT_REF_C(
      MAP_MAP *M,
      ICS_CHAR *NAME,
      OBJ_Object VALUE
)
```

Description: Identical to `MAP_INSERT_OBJECT_C()` except that only a reference to the `VALUE` object is stored in the map. Use this function in performance critical code sections, but be aware of possible side effects.

Example: `MAP_INSERT_OBJECT_REF_C(&M,(ICS_CHAR *)"T",OBJ);`

Function:

```
ICS_VOID MAP_MAP_VALUE_C(
      MAP_MAP M,
      ICS_CHAR *NAME,
      STRNG_STRING *VALUE,
      ICS_BOOLEAN *OK
)
```

Description: Finds the string value `VALUE` of the component `NAME` in the map `M`. If the component exists `OK` is `ICS_TRUE`, else `ICS_FALSE`. Use this function only when the value is a string, otherwise use `MAP_MAP_OBJECT_C()` to retrieve values from maps.

Example: `MAP_MAP_OBJECT_C(M,(ICS_CHAR *)"T",&VALUE,&OK)`

Function:

```
ICS_VOID MAP_MAP_OBJECT_C(
      MAP_MAP M,
      ICS_CHAR *NAME,
      OBJ_Object *VALUE,
      ICS_BOOLEAN *OK
)
```

Description: Finds the value `VALUE` of the component `NAME` in the map `M`. If the component exists `OK` is `ICS_TRUE`, else `ICS_FALSE`. `VALUE` is a full copy of the object stored in the map. If you don't need a full copy use `MAP_MAP_OBJECT_REF_C()` instead.

Example: `MAP_MAP_OBJECT_C(M,(ICS_CHAR *)"T",&VALUE,&OK)`

Function:

```
ICS_VOID MAP_MAP_OBJECT_REF_C(
      MAP_MAP M,
      ICS_CHAR *NAME,
      OBJ_Object *VALUE,
      ICS_BOOLEAN *OK
)
```

Description: Identical to `MAP_MAP_OBJECT_C()`, except that only a reference to the object value is returned as the result. Use this function if you need maximum performance but be aware of possible side effects.

Example: `MAP_MAP_OBJECT_REF_C(M,(ICS_CHAR *)"T",&VALUE,&OK)`

Function:

```
ICS_VOID MAP_REMOVE_OBJECT_C(MAP_MAP *M,
      ICS_CHAR *NAME
      ICS_BOOLEAN *OK
)
```

Description: Removes the component with name `NAME` from the map `M`. If the operation was successful `OK` is `ICS_TRUE`, else `ICS_FALSE`.

Function:

```
ICS_VOID MAP_CLEAR(
      MAP_MAP *M
)
```

Description: Removes all components from the map `M`.

Function:

```
ICS_VOID MAP_RESET_READ(
      MAP_MAP *M
)
```

Description: Resets the map `M` for read operations.

Function:

```
ICS_VOID MAP_READ_OBJECT(
      MAP_MAP *M,
      STRNG_STRING *NAME,
      OBJ_Object *OBJ,
      ICS_BOOLEAN *OK
)
```

Description: Reads the next name/value pair from the map `M`. A full copy the value is returned in `OBJ`.

Function:

```
ICS_VOID MAP_READ_OBJECT_REF(
      MAP_MAP *M,
      STRNG_STRING *NAME,
      OBJ_Object *OBJ,
      ICS_BOOLEAN *OK
)
```

Description: Reads the next name/value pair from the map `M`. A reference to the value is returned in `OBJ`.

Function:
```
ICS_VOID MAP_REMOVE(
      MAP_MAP * M
)
```
Example: Removes the local map `M`.

Function:
```
ICS_VOID MAP_COPY(
      MAP_MAP M1,
      MAP_MAP *M2
)
```
Example: Makes a full copy of map `M1` in map `M2`. `M2` is created first, then all values are copied from `M1` to `M2`.

6.4 Object Function Group (#include <object.h>, <btyp.h>)

Function:
```
ICS_VOID OBJ_init_objectvar(
      OBJ_Object *OBJ
)
```
Description: Initializes an object. Use this function always as the first operation on an object variable.

Function:
```
ICS_VOID OBJ_remove_object(
      OBJ_Object *OBJ
)
```
Description: Removes an object. The object variable is initialized again.

Function:
```
ICS_INT OBJ_get_type(
      OBJ_Object OBJ
)
```
Description: Returns the object TID as an integer.

Example:
```
if (OBJ_get_type(&OBJ) == BTYP_int_tidf()) {
      MESSAGE_DISPLAY_C((ICS_CHAR *)"integer");
}
```

Function:
```
ICS_VOID OBJ_copy_object(
      OBJ_Object OBJ1,
      OBJ_Object *OBJ2
)
```
Description: Copies object `OBJ1` to `OBJ2`.

Function:
```
ICS_VOID OBJ_display_object(
      OBJ_Object OBJ
)
```
Description: Displays the value of object `OBJ`. Use this function for debugging.

Function:
```
ICS_BOOLEAN OBJ_object_value_is_defined(
      OBJ_Object OBJ
)
```
Description: If the value of `OBJ` is defined this function returns `ICS_TRUE`, else `ICS_FALSE`.

Example:
```
if (OBJ_object_value_is_defined(OBJ) == ICS_TRUE) {
      MESSAGE_DISPLAY_C("value is defined\n");
}
```

Function:
```
ICS_INT BTYP_list_tidf(
)
```
Description: Returns the `TID` of type list.

Function:
```
ICS_VOID BTYP_create_list(
      OBJ_Object *OBJ
)
```
Description: Creates an object of type list. The created list is empty.

Function:
```
ICS_VOID BTYP_create_undef_list(
      OBJ_Object *OBJ
)
```
Description: Creates an object of type list. The object value ist set to undefined.

Function:
```
ICS_VOID BTYP_append_to_list(
      OBJ_Object l,
      OBJ_Object obj
)
```
Description: Appends object <obj> to list <l>.

Function:
```
ICS_VOID BTYP_reset_read_list(
      OBJ_Object l
)
```
Description: Resets the read pointer of list <l>.

Function:
```
ICS_VOID BTYP_read_next_listobj_ref(
      OBJ_Object l,
      OBJ_Object *obj,
      ICS_BOOLEAN *ok
)
```
Description: Returns a Pointer to the next object in list <l>. If no next object is available <ok> is set to `ICS_FALSE`.

Function:
```
ICS_INT BTYP_int_tidf(
)
```
Description: Returns the `TID` of type integer.

Function:
```
ICS_VOID BTYP_create_int(
      OBJ_Object *OBJ,
      ICS_INT32 IVAL
)
```
Description: Creates an object of type integer. The object value is initialized with `IVAL`.

Function:
```
ICS_VOID BTYP_create_undef_int(
      OBJ_Object *OBJ
)
```
Description: Creates an object of type integer. The object value is set to undefined.

Function:
```
ICS_VOID BTYP_write_int(
      OBJ_Object *OBJ,
      ICS_INT32 IVAL
)
```
Description: Overwrites the value of an integer object.

Function:
```
ICS_VOID BTYP_read_int(
      OBJ_Object *OBJ,
      ICS_INT32 *IVAL
)
```
Description: Reads the value of an integer object.

Function:
```
ICS_INT BTYP_real_tidf(
)
```
Description: Returns the TID of type real.

Function:
```
ICS_VOID BTYP_create_real(
      OBJ_Object *OBJ,
      ICS_REAL64 RVAL
)
```
Description: Creates an object of type real. The object value is initialized with `RVAL`.

Function:
```
ICS_VOID BTYP_create_undef_real(
      OBJ_Object *OBJ
)
```
Description: Creates an object of type real. The object value is set to undefined.

Function:
```
ICS_VOID BTYP_write_real(
      OBJ_Object *OBJ,
      ICS_REAL64 RVAL
)
```
Description: Overwrites the value of a real object.

Function:
```
ICS_VOID BTYP_read_real(
      OBJ_Object OBJ,
      ICS_REAL64 *RVAL
)
```
Description: Reads the value of a real object.

Function:
```
ICS_INT BTYP_boolean_tidf(
)
```
Description: Returns the TID of type boolean.

Function:
```
ICS_VOID BTYP_create_boolean(
      OBJ_Object *OBJ,
      ICS_BOOLEAN BVAL
)
```
Description: Creates an object of type boolean. The object value is initialized with `BVAL`.

Function:
```
ICS_VOID BTYP_create_undef_boolean(
      OBJ_Object *OBJ
)
```
Description: Creates an object of type boolean. The object value is set to undefined.

Function:
```
ICS_VOID BTYP_write_boolean(
      OBJ_Object *OBJ,
      ICS_BOOLEAN BVAL
)
```
Description: Overwrites the value of a boolean object.

Function:
```
ICS_VOID BTYP_read_boolean(
      OBJ_Object OBJ,
      ICS_BOOLEAN *BVAL
)
```
Description: Reads the value of a boolean object.

Function:
```
ICS_INT BTYP_string_tidf(
)
```
Description: Returns the TID of type string.

Function:
```
ICS_VOID BTYP_create_string(
      OBJ_Object *OBJ,
      STRNG_STRING SVAL
)
```
Description: Creates an object of type string. The object value is initialized with `SVAL`.

Function:
```
ICS_VOID BTYP_create_undef_string(
      OBJ_Object *OBJ
)
```
Description: Creates an object of type string. The object value is set to undefined.

Function:
```
ICS_VOID BTYP_write_string(
      OBJ_Object *OBJ,
      STRNG_STRING SVAL
)
```
Description: Overwrites the value of a string object.

Function:
```
ICS_VOID BTYP_read_string(
      OBJ_Object OBJ,
      STRNG_STRING *SVAL
)
```
Description: Reads the value of a string object.

Function:
```
ICS_INT BTYP_geometry_tidf(
)
```
Description: Returns the TID of type geometry.

Function:
```
ICS_VOID BTYP_create_geometry(
      OBJ_Object *OBJ,
      GEOM_GEOMETRY GVAL
)
```

Description: Creates an object of type geometry. The object value is initialized with `GVAL`. The object value is a full copy of `GVAL`.

Function:
```
ICS_VOID BTYP_create_geometry_ref(
      OBJ_Object *OBJ,
      GEOM_GEOMETRY GVAL
)
```
Description: Creates an object of type geometry. The object value is initialized with `GVAL`. Only a pointer to `GVAL` is stored in the object.

Function:
```
ICS_VOID BTYP_create_undef_geometry(
      OBJ_Object *OBJ
)
```
Description: Creates an object of type geometry. The object value is set to undefined.

Function:
```
ICS_VOID BTYP_write_geometry(
      OBJ_Object *OBJ,
      GEOM_GEOMETRY GVAL
)
```
Description: Overwrites the value of a geometry object.

Function:
```
ICS_VOID BTYP_read_geometry(
      OBJ_Object OBJ,
      GEOM_GEOMETRY *GVAL
)
```
Description: Reads the value of a geometry object. GVAL is a full copy of the object value.

Function:
```
ICS_VOID BTYP_read_geometry_ref(
      OBJ_Object OBJ,
      GEOM_GEOMETRY *GVAL
)
```
Description: Reads the value of a geometry object. `GVAL` is only a pointer to the object value.

Function:
```
ICS_INT BTYP_map_tidf(
)
```
Description: Returns the TID of type map.

Function:
```
ICS_VOID BTYP_create_map(
      OBJ_Object *OBJ,
      MAP_MAP MVAL
)
```
Description: Creates an object of type map. The object value is initialized with `MVAL`. The object value is a full copy of `MVAL`.

Function:
```
ICS_VOID BTYP_create_map_ref(
      OBJ_Object *OBJ,
      MAP_MAP MVAL
)
```
Description: Creates an object of type map. The object value is initialized with `MVAL`. Only a pointer to `MVAL` is stored in `OBJ`.

Function:
```
ICS_VOID BTYP_create_undef_map(
      OBJ_Object *OBJ
)
```
Description: Creates an object of type map. The object value is set to undefined.

Function:
```
ICS_VOID BTYP_write_map(
      OBJ_Object *OBJ,
      MAP_MAP MVAL
)
```
Description: Overwrites the value of a map object.

Function:
```
ICS_VOID BTYP_read_map(
      OBJ_Object OBJ,
      MAP_MAP *MVAL
)
```
Description: Reads the value of a map object. `MVAL` is a full copy of the object value.

Function:
```
ICS_VOID BTYP_read_map_ref(
      OBJ_Object OBJ,
      MAP_MAP *MVAL
)
```
Description: Reads the value of a map object. `MVAL` is only a pointer to the object value.

6.5 ICS Stack Function Group (#include <*icscpu.h*>)

Function:
```
ICS_VOID ICSCPU_PUSHI(
      ICS_INT32 VALUE
);
```
Description: Pushes an integer value on the ICS stack. Use this function to communicate with iG/Script programs.

Example: `ICSCPU_PUSHI((ICS_INT32)77);`

Function:
```
ICS_VOID ICSCPU_POPI(
      ICS_INT32 *VALUE
);
```
Description: Pops an Integer from the ICS stack. Use this function to communicate with iG/Script programs.

Function:
```
ICS_VOID ICSCPU_PUSHR(
      ICS_REAL64 VALUE
);
```
Description: Pushes a real value on the ICS stack. Use this function to communicate with iG/Script programs.

Example: `ICSCPU_PUSHR((ICS_REAL64)0.0);`

Function:
```
ICS_VOID ICSCPU_POPR(
      ICS_REAL64 *VALUE
);
```

Description: Pops a real value from the ICS stack. Use this function to communicate with iG/Script programs.

Function:
```
ICS_VOID ICSCPU_PUSHB(
      ICS_BOOLEAN VALUE
);
```
Description: Pushes a boolean value on the ICS stack. Use this function to communicate with iG/Script programs.

Example:
```
ICSCPU_PUSHB(ICS_TRUE);
```

Function:
```
ICS_VOID ICSCPU_POPB(
      ICS_BOOLEAN *VALUE
);
```
Description: Pops a boolean value from the ICS stack. Use this function to communicate with iG/Script programs.

Function:
```
ICS_VOID ICSCPU_PUSHS_C(
      ICS_CHAR *VALUE
);
```
Description: Pushes a string on the ICS stack. Use this function to communicate with iG/Script programs.

Example:
```
ICSCPU_PUSHS_C((ICS_CHAR*)"TEST");
```

Function:
```
ICS_VOID ICSCPU_POPS_C(
      ICS_CHAR *VALUE
);
```
Description: Pops a string from the ICS stack. Use this function to communicate with iG/Script programs.

Function:
```
ICS_VOID ICSCPU_PUSHM(
      MAP_MAP VALUE
);
```
Description: Pushes a full copy of the map VALUE on the ICS stack. Use this function to communicate with iG/Script programs.

Function:
```
ICS_VOID ICSCPU_POPM(
      MAP_MAP *VALUE
);
```
Description: Pops a map from the ICS stack. Use this function to communicate with iG/Script programs.

Function:
```
ICS_VOID ICSCPU_PUSH_OBJ(
      OBJ_Object OBJ
);
```
Description: Pushes a full copy of the object OBJ on the ICS stack. Use this function to communicate with iG/Script programs.

Function:
```
ICS_VOID ICSCPU_PUSH_OBJ_REF(
      OBJ_Object OBJ
);
```
Description: Pushes a reference of the object OBJ on the ICS stack. Use this function to communicate with iG/Script programs.

Function:
```
ICS_VOID ICSCPU_POP_OBJ(
      OBJ_Object *OBJ
);
```
Description: Pops an object from the ICS stack. Use this function to communicate with iG/Script programs.

Function:
```
ICS_VOID ICSCPU_POP(
);
```
Description: Deletes the topmost object from the ICS stack.

6.6 Display Function Group (#include <message.h>,<msg.h>)

Function:
```
ICS_VOID MESSAGE_DISPLAY_C(
      ICS_CHAR *FORMAT,
      ...
);
```
Description: Writes a message to the screen. The arguments are the same as for the standard C function `printf()`. You should always use `MESSAGE_DISPLAY_C()` instead of `printf()` because output of `MESSAGE_DISPLAY_C()` can be redirected by the kernel to a log file.

Example:
```
MESSAGE_DISPLAY_C((ICS_CHAR *)"count=%d\n",COUNT);
```

Function:
```
ICS_VOID MESSAGE_DISPLAY_ERROR_C(
      ICS_CHAR *FORMAT,
      ...
);
```
Description: Writes an error message to the screen. A kernel internal error counter is also inkremented. The arguments are the same as for the standard C function `printf()`. You should always use `MESSAGE_DISPLAY_ERROR_C()` instead of `printf()` because the output of `MESSAGE_DISPLAY_ERROR_C()` can be redirected by the kernel to a log file.

Example:
```
MESSAGE_DISPLAY_ERROR_C((ICS_CHAR *)"invalid file\n");
```

Function:
```
ICS_VOID MESSAGE_DISPLAY_STATUS_C(
      ICS_CHAR *FORMAT,
      ...
);
```
Description: Writes a status message to the screen. Some infoGrips products display a status line while running (iG/Import and iG/Export). You can set this status line with `MESSAGE_DISPLAY_STATUS_C()`. The arguments are the same as for the standard C function `printf()`. You should not terminate the string with a line feed character.

Example:
```
MESSAGE_DISPLAY_STATUS_C(
      (ICS_CHAR *)"now in topic Bodenbedeckung");
```

Function:
```
ICS_VOID MSG_CREATE_C(
      MSG_MSGPTR M *,
      ICS_CHAR *NAME,
      ICS_BOOLEAN *OK
);
```

Description: Creates a new message channel M with name NAME. The name is always displayed together with the actual message. Use one message channel per external module. The name of the message channel should be equal to name of the module. All MSG_* functions are implemented with MESSAGE_* functions.

Example: `MSG_CREATE_C(&M,(ICS_CHAR *)"CIN",&OK);`

Function:

```
ICS_VOID MSG_DISPLAY_C(
      MSG_MSGPTR M,
      ICS_CHAR *FORMAT,
      ...
);
```

Description: Displays a message on message channel M (see MESSAGE_DISPLAY_C() for more details). The function automatically adds a line feed character at the end of your message.

Example: `MSG_DISPLAY_C(M,"hello, world");`

Function:

```
ICS_VOID MSG_DISPLAY_ERROR_C(
      MSG_MSGPTR M,
      ICS_CHAR *FORMAT,
      ...
);
```

Description: Displays an error message on message channel M (see MESSAGE_DISPLAY_ERROR_C() for more details). The function automatically adds a line feed character at the end of your message.

Example: `MSG_DISPLAY_ERROR_C(M,(ICS_CHAR *)"wrong data type");`

Function:

```
ICS_VOID MSG_ABORT_C(
      MSG_MSGPTR M,
      ICS_CHAR *FORMAT,
      ...
);
```

Description: Displays an error message on message channel M and terminates the program (see MSG_DISPLAY_ERROR_C() for more details). Use this function to handle unrecoverable errors. After MSG_ABORT_C() has been called, the ICS kernel sends to all modules a <MODULE>_TERMINATE() request. Finally the kernel releases all memory used by objects. The standard C function exit() should never be used within an external module.

Example: `MSG_ABORT_C(M,(ICS_CHAR *)"division by zero");`

Function:

```
ICS_VOID MSG_SET_DSET_PROC(
      MSG_MSGPTR M *,
      ICS_BOOLEAN DEBUG
);
```

Description: If DEBUG is ICS_TRUE the channel M displays debug messages. Use this function to turn debugging messages for a channel on or off.

Function:

```
ICS_VOID MSG_SET_PROC_C(
      MSG_MSGPTR *M,
      ICS_CHAR *DEBUGMESSAGE
);
```

Description: If debugging is turned on for channel M the DEBUGMESSAGE is display on channel M.

Example: `MSG_SET_PROC_C(M,(ICS_CHAR *)"READ_OBJECT");`

6.7 Class Function Group (#include <*class.h*>)

Function:
```
ICS_VOID CLASS_CREATE_CLASS_C(
      ICS_CHAR *NAME,
      ICS_CHAR *VERSION,
      ICS_BOOLEAN *OK
);
```
Description: Creates a new class `NAME` with version `VERSION`.

Example:
```
CLASS_CREATE_CLASS_C((ICS_CHAR *)"CIN",
      (ICS_CHAR *)"1.0.0",&OK);
```

Function:
```
ICS_VOID CLASS_CREATE_METHOD_C(
      ICS_CHAR *CLASSNAME,
      ICS_CHAR *METHODNAME,
      CLASS_MPROC METHOD,
      ICS_INT16 IP,
      ICS_INT16 OP,
      ICS_CHAR MTYPE,
      ICS_CHAR *COMMENT,
      ICS_BOOLEAN *OK
);
```
Description: Creates a Method `METHODNAME` in the class `CLASSNAME`. The class has to exist already. `METHOD` is a C function without parameters (`void func()`), which implements the method. In `IP` you specify the number of arguments on the stack expected by the method. In `OP` you define the number of returned objects on the stack. `MTYPE` should be always set to `'P'`. In `COMMENT` you can describe your method. `OK` is `ICS_TRUE` if the operation was successful.

Example:
```
CLASS_CREATE_METHOD_C((ICS_CHAR *)"CIN",
      (ICS_CHAR *)"READ_OBJEKT",
      read_object,
      (ICS_INT16)0,(ICS_INT16)1,(ICS_CHAR)'P',
      (ICS_CHAR *)"READ_OBJECT[][b status]",&OK);
```

Function:
```
ICS_VOID CLASS_REMOVE_CLASS_C(
      ICS_CHAR *NAME,
      ICS_BOOLEAN *OK
);
```
Description: Deletes the class with name `NAME`. All methods within that class are also removed.

Example: `CLASS_REMOVE_CLASS_C((ICS_CHAR *)"CIN", &OK);`

6.8 Miscellaneous (#include <icslib.h>, <icsproc.h>, <icscpu.h>, <license.h>)

Function:
```
ICS_VOID ICSLIB_LOAD(
      ICS_CHAR *MNAME,
      ICS_CHAR *LNAME
);
```

Description:	Attaches the ICS kernel library to an external module. `MNAME` is the name of the external module. `LNAME` is the name of ICS kernel library. Use this function before you call any other function from the ICS library.

Function:
```
ICS_VOID ICSLIB_UNLOAD(
);
```
Description: Detaches the ICS kernel library from an external module. This should be the last function call to the ICS library in your external module.

Function:
```
ICS_VOID ICSPROC_COUNT_INPUT_OBJECT(
);
```
Example: Increments the internal input object counter by 1.

Function:
```
ICS_VOID ICSPROC_COUNT_OUTPUT_OBJECT(
);
```
Example: Increments the internal output object counter by 1.

Function:
```
ICS_VOID ICSCPU_DO_METHOD_BY_NAME_C(
      ICS_CHAR *CLASS,
      ICS_CHAR *METHOD
);
```
Description: Executes the ICS method `<CLASS>.<METHOD>`. Method arguments have to be pushed first on the ICS stack by `ICSCPU_PUSH_*` function calls. After method completion results can be accessed by `ICSCPU_POP_*` function calls. If the method does not exist this is a fatal error and the program is aborted.

Function:
```
ICS_INT LICENSE_get_module_license(
      ICS_CHAR *MODULE
);
```
Description: Checks the presence of a module license for module <MODULE>. If the license is valid, `LICENSE_FULL` or `LICENSE_DEMO` is returned as a result. If the license is invalid ICS is aborted with an error message.

Appendix

A1. References

[1] infoGrips GmbH. iG/Script Benutzer- und Referenzhandbuch, 1996

[2] Brian W. Kernighan, Dennis M. Ritchie. The Programming Language C, Second Edition, 1988

[3] infoGrips GmbH. ICS Benutzerhandbuch, 1997

A2. Module CIN

```
/************************************************************/
/* cin.c                                                    */
/* (c) 1996 by infoGrips GmbH, Zuerich                      */
/*                                                          */
/* This module implements an ICS input module               */
/*                                                          */
/* See also ICS Developmentsystem 2.1                       */
/*                                                          */
/************************************************************/

/* include files */
#include "icsrts.h"
#include "icslib.h"
#include "icsproc.h"
#include "strng.h"
#include "strng2.h"
#include "msg.h"
#include "class.h"
#include "icscpu.h"
#include "map.h"
#define MODULE_BODY
#include "cin.h"

/* global variables */
static STRNG_STRING s1;
static MSG_MSGPTR cin_msg;
static FILE *input_file;
static MAP_MAP cin_obj;

/* local functions */
static void open();
static void close();
static void read_object();
static void strip_newline();

/* init function of module CIN */
ICS_VOID DLLEXPORT CIN_INIT(mainlib)
  ICS_CHAR *mainlib;
{

  ICS_BOOLEAN ok;
```

```
    /* attach to ICS library */
    if (ICSLIB_LOAD("cin.dll",mainlib) == ICS_FALSE) {
       return;
    }

    /* create a message channel CIN */
    MSG_CREATE_C(&cin_msg, (ICS_CHAR *)"CIN", ICS_FALSE);

    /* create a class CIN with version 1.0.0 */
    CLASS_CREATE_CLASS_C((ICS_CHAR *)"CIN", (ICS_CHAR *)"1.0.0", &ok);

    /* register the methods OPEN, CLOSE and READ_OBJECT */

    /* method OPEN[s filename][] */
    CLASS_CREATE_METHOD_C((ICS_CHAR *)"CIN", (ICS_CHAR *)"OPEN", open,
      1, 0, 'P', "", &ok);
    /* method CLOSE[][] */
    CLASS_CREATE_METHOD_C((ICS_CHAR *)"CIN", (ICS_CHAR *)"CLOSE", close,
      0, 0, 'P', "", &ok);
    /* method READ_OBJECT[][b status] */
    CLASS_CREATE_METHOD_C((ICS_CHAR *)"CIN", (ICS_CHAR *)"READ_OBJECT",
  read_object,
      0, 1, 'P', "", &ok);

    /* initialize global strings */
    STRNG_CREATE(&s1);

  }

  /* terminate function of module CIN */
  ICS_VOID DLLEXPORT CIN_TERMINATE()
  {
    ICS_BOOLEAN ok;

    /* deallocate global strings */
    STRNG_REMOVE(&s1);

    /* delete class CIN */
    CLASS_REMOVE_CLASS_C((ICS_CHAR *)"CIN",&ok);

    /* detach from ICS library */
    ICSLIB_UNLOAD();

  }

  /* implementation of method CIN.OPEN */
  static ICS_VOID open()
  {

    ICS_BOOLEAN ok;
    MAP_MAP m;
    char    file_name[255];

    /* search map IN */
    MAP_GET_MAP_BY_NAME_C(&cin_obj, (ICS_CHAR *)"IN", &ok);
    if (ok == ICS_FALSE)
      MSG_ABORT_C(cin_msg, "map IN is not defined");

    /* search map CIN_PARAM */
    MAP_GET_MAP_BY_NAME_C(&m, (ICS_CHAR *)"CIN_PARAM", &ok);
    if (ok == ICS_FALSE)
      MSG_ABORT_C(cin_msg, (ICS_CHAR *)"map CIN_PARAM is not defined");
  /* test parameter CIN_PARAM.DEBUG */
  MAP_MAP_VALUE_C(m, (ICS_CHAR *)"DEBUG", &s1, &ok);
  if (ok) {
    if (STRNG2_MATCHA_C(s1, "ON")) {
        /* if CIN_PARAM.DEBUG = ON, turn debugging messages on */
        /* for channel CIN */
      MSG_SET_DSET_PROC(&cin_msg, ICS_TRUE);
    }
    else {
        /* else turn debugging messages off */
      MSG_SET_DSET_PROC(&cin_msg, ICS_FALSE);
      }
  }

  /* display debug message */
  MSG_SET_PROC_C(cin_msg, (ICS_CHAR *)"OPEN");

  /* get file name from ICS stack */
  ICSCPU_POPS(&s1);
  /* convert kernel string to C string */
  STRNG2_MSTRING_TO_CSTRING(&s1,file_name);

  /* open the file */
  input_file = fopen(file_name,"r");
  if (input_file == NULL) {
     MSG_ABORT_C(cin_msg,(ICS_CHAR *)"unable to open input file
%s",file_name);
  }

}

/* implementation of method CIN.CLOSE */
static void close()
{
   /* display debug message */
   MSG_SET_PROC_C(cin_msg, (ICS_CHAR *)"CLOSE");

   /* close the input file */
   fclose(input_file);
}

/* implementation of method CIN.READ_OBJECT */
static ICS_VOID read_object()
{
   char line[255];

   /* debugging message */
   MSG_SET_PROC_C(cin_msg, (ICS_CHAR *)"READ_OBJECT");

   /* clear map IN */
   MAP_CLEAR(&cin_obj);

   /* read a text line from the input file */
   if (fgets(line,255,input_file) == NULL) {
      /* return ICS_FALSE on the ICS stack if the method fails */
        ICSCPU_PUSHB(ICS_FALSE);
   }
   else {

     /* insert text line in component IN.TXT */
        strip_newline(line);
        MAP_INSERT_VALUE_C(cin_obj, (ICS_CHAR *)"TEXT", line);
      /* push ICS_TRUE on the ICS_STACK */
         ICSCPU_PUSHB(ICS_TRUE);

         /* increment the input counter */
      ICSPROC_COUNT_INPUT_OBJECT();

   }

}

/* miscalleanous functions */

/* strip new line character from a C string */
static ICS_VOID strip_newline(s)
   char *s;
{
   unsigned int i;
   i=0;
   while (i<strlen(s)) {
      if (s[i]=='\n') {
           s[i]='\0';
             return;
        }
      i++;
   }

}
```

A3. Module COUT

```
/************************************************************/
/* cout.c                                                   */
/* (c) 1996 by infoGrips GmbH, Zürich                       */
/*                                                          */
/* this module implements an ICS output module              */
/*                                                          */
/* see also ICS Development System 2.1                      */
/*                                                          */
/************************************************************/

/* include files */
#include "icsrts.h"
#include "icslib.h"
#include "icsproc.h"
#include "strng.h"
#include "strng2.h"
#include "msg.h"
#include "class.h"
#include "icscpu.h"
#include "map.h"
#define MODULE_BODY
#include "cout.h"

/* global variables */
static STRNG_STRING s1;
static MSG_MSGPTR cout_msg;
static FILE *output_file;
static MAP_MAP cout_obj;

/* functions */
static void open();
static void close();
static void write_object();

/* init function for module COUT */
ICS_VOID DLLEXPORT COUT_INIT(mainlib)
  ICS_CHAR *mainlib;
{

  ICS_BOOLEAN ok;

  /* attach to ICS library */
  if (ICSLIB_LOAD("cout.dll",mainlib) == ICS_FALSE) {
     return;
  }

  /* create a message channel COUT */
  MSG_CREATE_C(&cout_msg, (ICS_CHAR *)"COUT", ICS_FALSE);

  /* create class COUT version 1.0.0 */
  CLASS_CREATE_CLASS_C((ICS_CHAR *)"COUT", (ICS_CHAR *)"1.0.0", &ok);

  /* register methods OPEN, CLOSE and WRITE_OBJECT in class COUT */

  /* method OPEN[s filename][] */
  CLASS_CREATE_METHOD_C((ICS_CHAR *)"COUT", (ICS_CHAR *)"OPEN", open,
    1, 0, 'P', "", &ok);
  /* method CLOSE[][] */
  CLASS_CREATE_METHOD_C((ICS_CHAR *)"COUT", (ICS_CHAR *)"CLOSE",
close,
    0, 0, 'P', "", &ok);
  /* method READ_OBJECT[][b status] */
  CLASS_CREATE_METHOD_C((ICS_CHAR *)"COUT",
    (ICS_CHAR *)"WRITE_OBJECT", write_object, 0, 1, 'P', "", &ok);

  /* initialize global string variables */
  STRNG_CREATE(&s1);

}

/* terminate module COUT */
ICS_VOID DLLEXPORT COUT_TERMINATE()
{

  ICS_BOOLEAN ok;

  /* deallocate global string variables */
  STRNG_REMOVE(&s1);

  /* delete class COUT */
  CLASS_REMOVE_CLASS_C((ICS_CHAR *)"COUT",&ok);

  /* detach ICS library */
  ICSLIB_UNLOAD();

}


/* implementation of method COUT.OPEN */
static void open()
{

  ICS_BOOLEAN ok;
  MAP_MAP m;
  char    file_name[255];
```

```
    /* search map OUT */
    MAP_GET_MAP_BY_NAME_C(&cout_obj, (ICS_CHAR *)"OUT", &ok);
    if (ok == ICS_FALSE)
      MSG_ABORT_C(cout_msg, "map OUT is not defined");

    /* search map COUT_PARAM */
    MAP_GET_MAP_BY_NAME_C(&m, (ICS_CHAR *)"COUT_PARAM", &ok);
    if (ok == ICS_FALSE)
      MSG_ABORT_C(cout_msg, (ICS_CHAR *)"map COUT_PARAM is not de-
  fined");

    /* is parameter COUT_PARAM.DEBUG = ON ? */
    MAP_MAP_VALUE_C(m, (ICS_CHAR *)"DEBUG", &s1, &ok);
    if (ok) {
      if (STRNG2_MATCHA_C(s1, "ON")) {
          /* turn on debugging messages for channel COUT */
        MSG_SET_DSET_PROC(&cout_msg, ICS_TRUE);
      }
      else {
          /* turn off debugging messages for channel COUT */
        MSG_SET_DSET_PROC(&cout_msg, ICS_FALSE);
        }
    }

    /* display debugging message */
    MSG_SET_PROC_C(cout_msg, (ICS_CHAR *)"OPEN");

    /* get file name from ICS stack */
    ICSCPU_POPS(&s1);
    /* convert kernel string to C string */
    STRNG2_MSTRING_TO_CSTRING(&s1,file_name);

    /* create the file */
    output_file = fopen(file_name,"w");
    if (output_file == NULL) {
       MSG_ABORT_C(cout_msg,(ICS_CHAR *)"unable to create output file
  %s",file_name);
    }

  }

  /* implementation of method COUT.CLOSE */
  static void close()
  {
     /* display debugging message */
     MSG_SET_PROC_C(cout_msg, (ICS_CHAR *)"CLOSE");

     /* close output file */
     fclose(output_file);
  }

  /* implementation of method COUT.WRITE_OBJECT */
  static void write_object()
  {
     ICS_BOOLEAN ok;
     char line[255];

     /* display debugging message */
     MSG_SET_PROC_C(cout_msg, (ICS_CHAR *)"WRITE_OBJECT");

     /* read text line from OUT.TXT */
     MAP_MAP_VALUE_C(cout_obj,(ICS_CHAR *)"TEXT",&s1,&ok);
```

```
   if (ok == ICS_FALSE) {
      MSG_ABORT_C(cout_msg,"component OUT.TEXT is not defined");
   }

   /* convert kernel string to C string */
   STRNG2_MSTRING_TO_CSTRING(&s1,line);
   /* write the text line to the output file */
   fprintf(output_file,"%s\n",line);

   /* increment the output counter */
   ICSPROC_COUNT_OUTPUT_OBJECT();

}
```

A4. Test Scripts

A4.1 test1.cfg

```
! test1.cfg
!
! This iG/Script program writes with COUT
!
!    hello, World
!    this is a test
!
! to the file ILTOOLS_DIR\data\test.dat.
!
! Debugging messages are turned on.

|LICENSE \license\icsdev.lic
|LOAD cout

MAP COUT_PARAM
   DEBUG => ON
END_MAP

DISPLAY 'ICS-TEST 1'

OPT.ics_dir . '\data\test.dat' COUT.OPEN

'hello, World' => OUT.TEXT
COUT.WRITE_OBJECT

'this is a test' => OUT.TEXT
COUT.WRITE_OBJECT

COUT.CLOSE
```

A4.2 test2.cfg

```
! test2.cfg
!
! This script reads with CIN the file ILTOOLS_DIR\data\test.dat
! and displays it's contents in the logfile.
!
! Debugging messages are turned on.
!

|LICENSE \license\icsdev.lic
|LOAD cin
```

```
MAP CIN_PARAM
   DEBUG => ON
END_MAP

DISPLAY 'ICS-TEST 2'

OPT.ics_dir . '\data\test.dat' CIN.OPEN

WHILE CIN.READ_OBJECT DO
   DISPLAY IN.TEXT
END_WHILE

CIN.CLOSE
```

A5. Configuration copy.cfg

```
! copy.cfg
!
! This iG/Script configuration copies textfiles with
! the modules CIN and COUT.
!
! Debugging messages are turned on.
!

|LICENSE \license\icsdev.lic
|LOAD cin,cout,dialog

MAP CIN_PARAM
   DEBUG => ON
END_MAP

MAP COUT_PARAM
   DEBUG => ON
END_MAP

DISPLAY 'ICS-COPY'

IF 'OPT.input' EXISTS NOT THEN
   IF 'Enter Input File' 'dat' FALSE DIALOG.GET_FILENAME THEN
      => OPT.input
   ELSE
      DISPLAY 'script aborted by user'
      HALT
   END_IF
END_IF

IF 'OPT.output' EXISTS NOT THEN
   IF 'Enter Output File' 'dat' FALSE DIALOG.GET_FILENAME THEN
      => OPT.output
   ELSE
      DISPLAY 'script aborted by user'
      HALT
   END_IF
END_IF

OPT.input  CIN.OPEN
OPT.output COUT.OPEN

WHILE CIN.READ_OBJECT DO
   IN.TEXT => OUT.TEXT
   COUT.WRITE_OBJECT
END_WHILE
```

```
CIN.CLOSE
COUT.CLOSE
```