

# **Handcoder's Guide to OCS Version 2**

---

A (DRAFT) manual for the brave

**The OPAL Group**  
written by **Wolfgang Grieskamp and Mario Suedholt**

---

Copyright © 1994 by The OPAL Group

# 1 Introduction

This manual describes how to add *handcoded structures* to the OPAL Compilation System Version 2.

## 1.1 What's the use of Handcoding

OPAL is a *scheme* language, by which we understand that it doesn't provide a priori more than the essential builtin data types of booleans and textual denotations. Consequently, the OPAL compiler must support a way to add handcoded structures to the compilation system without loosing efficiency compared to as data types defined by these structures are built into the language.

There is a second — even more important — reason which is independent of OPAL being a scheme language. In most cases software is not constructed from scratch, but is embedded in and does embed existing software: the operating system, the user interface, the data base manager and so on. This calls for a well defined *interlanguage working* interface. OPAL programs must have the possibility to access existing software and must themselves be accessible from other software. Nowadays C is the de facto standard for system programming; hence, it is desirable that OPAL programs can incorporate with C software and vice versa.

## 1.2 Target and Foreign Language

The OPAL compilation system Version 2 uses ANSI C as its target language. The decision to use C has been driven by the demand to produce highly platform-independent code, while preserving some access to machine oriented features like pointer calculations. Although this approach is not totally satisfactory, since for example the generated C code is (redundantly) parsed and context-checked by the C compiler, we expect that it will survive some time.

So we are in the happy situation that the target language — to be used for handcoding — and the foreign language — to be used to connect to existing software — fall together. Nevertheless, the handcoding scheme documented in this manual is well prepared for future changes of the target language. Only those features crucial for efficiency of handcoded low-level data types will change if the target language changes; those are grouped mainly around the inlining of function definitions via the C macro-preprocessor. Handcoded structures which avoid these features will probably port to future versions of OCS.

## 1.3 Warning

Before you plan to handcode an OPAL structure be aware that this is a job for the brave. You should be familiar with writing C programs as well as with writing OPAL programs. You should have some idea of how functional languages are compiled. On the level of handcoding the (weak) type discipline of C is completely broken; this is necessary to model the richer type system of OPAL. Face that you have a lot to do with tedious and tricky low-level memory management — stuff the compiler normally handles automatically for you. Be aware that debugging handcoded structures, in particular if they are concerned with

more complex data objects, is a boring task. The usual crash you have to expect occurs in some of the memory management functions of the runtime system, since a dangling pointer has been accessed, which has been introduced in completely different part of the program.

## 2 Handcoded Structures

A handcoded structure consists of four source parts, each kept in a separate file. The signature and implementation parts are ordinary OPAL documents. Additionally, there are a handcoded interface and a handcoded implementation part, both written in C. The handcoded source parts are combined internally with the C interface and implementation parts derived from the OPAL sources.

### 2.1 Source Parts of Handcoded Structures

The four source parts of a handcoded structure are:

#### *Struct.sign*

This is the OPAL signature part of a handcoded structure. It is in no way different from ordinary OPAL signatures.

#### *Struct.impl*

This file contains the OPAL implementation part of a handcoded structure. There is no fundamental difference to ordinary implementations, except:

- The global pragma `/$ handcoded $/` appears somewhere in the file.<sup>1</sup>
- Functions which shall be handcoded are not implemented here. Be aware that there is no notion of a function being *partially* implemented; in particular, leaving out cases in pattern based function definition leads to an implemented function which is undefined in some cases but can not be handcoded.
- Sorts which shall be handcoded are not implemented here either. However, pseudo implementations may be given in order to simulate a storage class of the sort for more efficient memory management. See Section 4.3.5 [Simulating Storage Classes], page 17.

#### *Struct.hc.h*

This is the handcoded interface part of the structure. It contains non-canonical definitions for use in other handcoded structures and canonical macro based implementations of functions, if such are supplied. See Section 5.3 [Macro Entries], page 21.

#### *Struct.hc.c*

This is the handcoded implementation part of the structure. For each n-ranked function not implemented in OPAL it contains a canonical implementation in C (see Section 5.2 [Direct Entries], page 21). For each zero-ranked function (constant) it contains canonical initialization code, activated through a special initialization function (see Section 5.5 [Constants], page 23). It furthermore may contain other non-canonical C code.

---

<sup>1</sup> This is not essential in the current version of OCS; incidently, the compiler will not issue a warning if it is lacking. It might, however, become necessary in future versions

## 2.2 A Look and Feel Example

Let us take a view at a simple handcoded structure. Not all information to understand this example in detail has been presented yet, but it might serve to give you a first taste.

As an example we use a simplified version of the structure `DEBUG` from the OPAL Standard Library, which provides a function for side-effect prints. The OPAL signature part looks as follows:

```
SIGNATURE DEBUG[data]
SORT data
FUN PRINT : bool**denotation**data->data
```

The OPAL implementation uses a handcoded internal function `print` to realize the side-effect print:

```
IMPLEMENTATION DEBUG
/$ handcoded $/
DEF PRINT(true,msg,data) == print(msg,data)
DEF PRINT(false,msg,data) == data
-- function to become handcoded:
FUN print: denotation ** data -> data
```

The handcoded interface part is empty in this example:

```
/* hand-coded interface part of DEBUG */
```

The handcoded implementation part makes use of the runtime system function `get_denotation`, which converts an OPAL object of sort `denotation` to a C string, and a global char buffer named `charbuf`

```
/* hand-coded implementation part of DEBUG */

#include <stdio.h>

extern OBJ _ADEBUG_Aprint(OBJ msg,OBJ data) /* print */ {
    fprintf(stderr,"DEBUG PRINT:\n");
    get_denotation(msg,charbuf,CHARBUFSIZE);
    fputs(stderr,charbuf);
    fputc('\n',stderr);
    return data;
}

static init_const_ADEBUG()
{}
```

## 2.3 Derived Parts of Handcoded Structures

The OPAL compiler generates C interface and implementation parts from the OPAL parts of a handcoded structure. These files are normally not of interest to the user, but it is useful to understand how the final C source is assembled from the generated and handcoded C parts.

`[OCS/] Struct.h`

This is the generated C interface part of the structure. It primarily declares all external canonical functions and constants, and defines *symbolic name aliases* for all objects (the compiler internally uses numerically encoded names, see Chapter 3 [Naming], page 8). It *includes* the handcoded interface part `Struct.hc.h`.

`[OCS/] Struct.c`

This is the generated C implementation part of the structure. It declares all internal canonical functions, constants and symbolic aliases, contains the generated code of those functions implemented in OPAL, and some additional administrative stuff. It *includes* the handcoded implementation part `Struct.hc.c`.

To continue the example from the last section (see Section 2.2 [Look and Feel], page 4), the derived C interface for `DEBUG` follows. We have added comments to illustrate the general scheme:

```
#ifndef ADEBUG_included
#define ADEBUG_included

/* Name aliases */
#define __ADEBUG_APRINT __ADEBUG_1
#define _ADEBUG_APRINT _ADEBUG_1
#define ADEBUG_1 ADEBUG_APRINT

/* Extern Declarations */
extern OBJ __ADEBUG_APRINT;
extern OBJ _ADEBUG_APRINT(OBJ,OBJ,OBJ);

/* Inclusion of handcoded part */
#include "DEBUG.hc.h"

/* Default Definition of Macro Implementations */
#ifndef ADEBUG_APRINT /* May be overwritten in hc.h part */
#define ADEBUG_PRINT(x1,x2,x3,r) {r=_ADEBUG_APRINT(x1,x2,x3);}
/* Map to function implementation */
#endif

#endif /* ADEBUG_included */
```

The derived C implementation is the following:

```
/* Inclusion of runtime system definitions */
#include "BUILTIN.h"

/* Inclusion of imported and own definitions */
#include "DEBUG.h"

/* Name Aliases for internal functions */
#define __ADEBUG_Aprint __ADEBUG_2
#define _ADEBUG_Aprint _ADEBUG_2
```

```

#define ADEBUG_2 ADEBUG_Aprint

/* Function Declarations and Closure Variable Definitions */
extern OBJ _ADEBUG_APRINT(OBJ,OBJ,OBJ); OBJ __ADEBUG_APRINT; /* PRINT */
extern OBJ _ADEBUG_Aprint(OBJ,OBJ,OBJ); OBJ __ADEBUG_Aprint; /* print */

/* Inclusion of handcoded implementation part */
#include "DEBUG.hc.c"

/* Generated C code for PRINT */
extern OBJ _ADEBUG_APRINT(OBJ x1,OBJ x2,OBJ x3) /* PRINT */
{...}

/* Closure Evaluation Entries */
....

/*
init_ADEBUG(){
    static int visited=0; if(visited) return; visited=1;
    ...
    init_const_ADEBUG();
}

```

## 2.4 Maintaining Handcoded Structures

The OPAL compiler is capable of generating *templates* for the handcoded parts of a structure. This is supported by the OCS drivers `ocs` resp. `ors`. The intended procedure is as follows:

1. Create a subsystem for handcoded structures, using the appropriate system description template (usually found under `ocs/lib/om/tmpls/SysDefs.subhc.tmpl`). Most of the settings to be filled in correspond to those of ordinary OPAL top-level or subsystem templates (See *Users Guide to OPAL*). Only the variables

### **NORMSTRUCTS**

Make Variable

This variable lists all ordinary OPAL structures, i.e. those without a hand-coded part.

and

### **FOREIGNSTRUCTS**

Make Variable

This variable lists all OPAL structures with handcoded definitions.

are specific to this template file.

Please note that you cannot maintain handcoded structures in a top-level system, nor can specify the system description from the command line of `ocs` or `ors` — you must use the system description template.

2. Edit the OPAL signature and implementation parts of the structure. Implement all functions you wish to code in OPAL. If you can not give the final definition of a



function, give at least a pseudo implementation (e.g. `DEF f(x) == f(x)`), since the compiler decides for which functions templates are generated on the basis whether a function is implemented or not.

3. Invoke `ocs` resp. `ors`. The OPAL parts of the handcoded structure will be compiled to the derived files (usually located in the `OCS` subdirectory) as well as to two files named `Struct.hc.h.tmp1` and `Struct.hc.c.tmp1`.
4. If you are compiling for the first time, `ocs` will stop with a message like ‘Don’t now how to make target `Struct.hc.c`’. This is perfectly okay, since `ocs` actually doesn’t know – its your job.

You usually now derive the handcoded parts from the templates. The template for the header is just empty. The template for the implementation file contains definitions for all unimplemented functions, with the bodys filled up by a `HLT` statement.

5. If you are not compiling the first time, the templates are created with each recompilation and ignored. If changes have been made, the handcoder is responsible to merge the changed templates to the handcoded structure parts.

## 2.5 Accessing Plain Structures from Handcoded Structures

Per default, the OPAL compiler does not generate interfaces to access plain structures from within handcoded structures. To instruct the compiler to do so, you must use the same system description file as for handcoded system (see Section 2.4 [Maintenance], page 6), and fill in the variable `NORMSTRUCTS` with those structures you wish to access from handcoded structures. For these structures the compiler generates derived interfaces `[OCS/]Struct.h`, which behave similar as for handcoded structures – except that the handcoded interface and implementation part is regarded to be empty, and thus not included.

## 3 Naming Conventions

OPAL functions are referred to by numeric names in the generated C code. To enable comfortable handcoding the derived C parts of a hancoded structure declare symbolic aliases for numeric names. However, the symbolic names cannot be one-to-one translations, since the lexical rules of OPAL and C are not compatible. Furthermore, the namespace of C programs is flat, which requires to augment names with the originating structure. Last not least overloading of names originating from the same structure has to be coped with.

### 3.1 Transliteration of OPAL Identifiers

A transliteration maps each valid OPAL identifier to a valid C identifier. The method is as follows.

1. First, each OPAL identifier is partitioned in fragments of either alphanumeric or special letters. This is the syntax for the fragmentation:

$$\begin{aligned} \text{Ident} &::= \text{Fragment} ( \text{'\_'} [ \text{Fragment} ] )^* \\ \text{Fragment} &::= \text{Letgit+ '?'*} \mid \text{Special+} \end{aligned}$$

(The character class *Letgit+* is defined in the report *The Programming Language OPAL*). Note that *'\_'* is itself a special letter. Ambiguity is resolved as follows: if the current fragment consists of specials, underlines are taken to be members of this fragment, except a trailing underline which introduces the next letgit fragment. If the identifier ends with a letgit fragment followed by a single *'\_'*, this is interpreted as the introduction of an empty trailing special fragment.

2. Next, each fragment is transliterated as follows: a letgit fragment is taken as is but with a prepended *'A'* and all trailing *'?'* substituted by *'\_'*. A special fragment is prepended by a *'S'*, and its characters are mapped according to the following table:

|      |   |   |   |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|------|---|---|---|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Opal | ! | @ | # | \$ | % | ^ | & | * | + |   | ~ | - | = | \ | ' | { | } | [ | ] | : | ; | < | > | . | / | ? | _ |
| C    | 1 | 2 | 3 | 4  | 5 | 6 | 7 | 8 | p | S | t | m | e | b | q | 0 | C | o | c | i | a | l | g | d | s | _ | u |

If you have a US keyboard, notify the correlation between the numeric keys and the special character associated with the according shifted key.

3. Last, the transliterated fragments are reconcatenated, separated by *'\_'*.

The following table shows some normal and some pathological examples of transliterated identifiers:

|           |              |
|-----------|--------------|
| DEBUG     | ADEBUG       |
| empty?    | Aempty_      |
| ::?       | Sii_         |
| ::?__     | Sii_uu       |
| ::?__foo_ | Sii_u_Afoo_S |

### 3.2 Basenames of Functions

Each OPAL function has an associated *basename*. The basename is used to generate the names of the different C objects associated with an OPAL function, see Chapter 5 [Functions], page 20. The basename is constructed as follows:

1. The transliterated structure identifier is concatenated with the transliterated function identifier, separated by ‘\_’.
2. If there are several overloaded function versions originating from the same structure, the suffix ‘\_01’ is appended to the name of the second version, ‘\_02’ to the name of the third version, and so on. The order of overloaded versions is defined by the textual order of the declarations in the OPAL handcoded parts.

**Caution:** The naming scheme for overloaded functions from the same structure is rather dangerous. You might add an overloaded function to an existing handcoded structure if it is declared textually after the current versions, or you might delete the textually last version. In all other cases you have to watch out for changed names and update your sources accordingly. If you make a mistake, there is no one who complains. *We recommend not to use overloaded names at all for handcoded functions.* You can always perform a simple renaming in the OPAL implementation part to avoid overloaded names in the handcoded part, for example:

```
FUN f: s -> s
FUN f: t -> t

-- rename to avoid overloading
FUN f_s: s -> s
FUN f_t: t -> t
DEF f == f_s
DEF f == f_t
-- f_t and f_s handcoded
```

## 4 Data Objects and Memory Management

One of the main differences between functional and imperative languages is the treatment of data objects: in functional languages the notions of memory and reference do not exist and data types are described by rather abstract and possible recursive definitions. In imperative languages the notion of memory and reference is most important, since dynamic objects can only be represented by pointer structures. Moreover, deallocation of memory which is not longer used has to be performed manually.

Clearly, the difference in the handling of data objects is the factor which complicates a powerful interlanguage working interface between functional and imperative languages.

In OCS, every OPAL data object is represented by a uniform C object of type OBJ. This uniformity is necessary to support parametric resp. polymorphic function and data type definitions. Usually, type OBJ is defined in C as `void *`, but you should treat it as an abstract entity.

An object of type OBJ either stands for a primitive, self-containing value, or is a reference to a structured heap cell. The first kind is called for short *primitive object*, the second kind *structured object*, which subsumes the reference as well as the heap cell.

Each value of type OBJ is equipped with a type tag in order to distinguish the two kinds.

OCS uses an enhanced lazy reference counting approach to memory management which results in a *residual* garbage collector, that is, the garbage collector is compiled online into the code. This makes hand-coding more tedious, since the handcoder also has to perform garbage collection within his code, as long as she calculates with OPAL data objects. More exactly, the handcoder is responsible for correctly maintaining the reference counter.

### 4.1 Primitive Objects

A primitive object stands for a self-containing value. To access the value the object must be *unpacked*, that is, the type tag must be removed. To create a primitive object the value must be *packed*, i.e. the type tag must be set. Be aware that passing around unpacked values as OBJs almost certainly crashes the program, since they may be interpreted as references by the runtime system.

The following macro tests whether an object is primitive:

```
int is_primitive (OBJ obj) Macro
    Returns 1, if obj is a primitive object, otherwise 0.
```

#### 4.1.1 Unsigned Integers as Primitive Objects

There are several functions to handle primitive objects which represent unsigned integers:

```
WORD Type
    The type of unsigned integers which are large enough to hold the value of an
    unpacked primitive object.

OBJ pack_word (WORD word) Macro
WORD unpack_word (OBJ obj) Macro
    Packs respectively unpacks a primitive object as an unsigned word.
```

**WORD max\_word** Constant  
 The maximal unsigned integer which may be packed in a primitive object. On a 32 Bit machine, this is usually  $2^{31}$ , since 1 bit is used for the type tag.

**WORD bits\_per\_word** Constant  
 The number of bits available in a **WORD**, usually 31.

### 4.1.2 Signed Integers as Primitive Objects

These are the functions to handle primitive objects which represent signed integers:

**SWORD** Type  
 The type of signed integers which is large enough to hold the value of an unpacked primitive object.

**OBJ pack\_sword** (**SWORD** *sword*) Macro  
**SWORD unpack\_sword** (**OBJ** *obj*) Macro  
 Packs respectively unpacks a primitive object as a signed word.

**SWORD min\_sword** Constant  
**SWORD max\_sword** Constant  
 The minimal and maximal unsigned value which may be packed in a primitive object.

### 4.1.3 Pointers as Primitive Objects

In some situations it might be useful to treat a pointer, returned by a C library function or explicitly created by **malloc**, as a primitive object. A typical situation are pointers of type **FILE\*** from the C standard library.

Pointers can be packed and unpacked:

**OBJ pack\_pointer** (**void\*** *ptr*) Macro  
**void\* unpack\_pointer** (**OBJ** *obj*) Macro  
 Packs respectively unpacks a pointer.

Be aware that these functions rely on pointers always pointing to even byte boundaries. The following macro symbol tells you whether this is *not* the case, and you must explicitly align pointers by yourself (or do not use this feature at all):

**\_ALIGN\_POINTERS\_** Symbol  
 If defined, the platform does not ensure that pointers returned by library functions or by **malloc** are aligned on even byte boundaries.

Currently, we are not aware of any UNIX architecture which requires this flag to be set.

Please note that memory allocated from the runtime system is always correctly aligned. See Section 4.5 [Auxiliary Memory], page 19.

## 4.2 Structured Objects

Structured objects are references pointing to heap cells. Structured objects in this sense encompass both the reference as well as the cell.

OCS distinguishes between *deep*, *flat*, and *byte-flat* structured objects. The cell of a deep object consists of other objects which are subject to garbage collection. A flat object consists of arbitrary data which is not subject to garbage collection. A byte-flat object may in most cases be identified with a flat object; an exception is persistent binary data storage and network interchange, where byte-flat objects have to be distinguished from normal flat objects to allow the exchange of byte streams between architectures with different endians.

Each of the three kinds of objects is furthermore partitioned into *small* and into *big* objects. Small objects are allocated and deallocated using an array of size-specific free lists; big objects share the same free list regardless of their size.

The kinds of structured objects are determined at allocation time. There are no possibility and no necessity to test object kinds. There is, however, one macro which generally tests whether an object is structured:

```
int is_structured (OBJ obj) Macro
    Returns 1, if obj is a structured object, otherwise 0.
```

### 4.2.1 Declaring Structured Data

The layout of a structured object is declared by a plain C **struct** declaration. As the first component, however, you have to include a header predefined in the runtime system.

There are two kind of headers, one for definitively small objects, and one for objects which may be either small or big. The first kind is normally used for objects of fixed size (which are not bigger than at most 127 words), the second for objects with dynamically growing size. Small objects require one word less of memory resources than big objects with respect to the overhead of memory management.

#### 4.2.1.1 Declaring Small Objects

The following header is used for declaring a small structured object:

```
struct sCELL Structure
    Include this structure as a first component of your data structure if it will be
    definitively small.
```

A typical example is the declaration of reals:

```
typedef struct sREAL {
    struct sCELL header;
    double value;
} * REAL;
```

Please note that this must become a flat object, since the double value should not be subject to garbage collection. However, the flatness of a structured object is determined at allocation time, see Section 4.2.2 [Allocating], page 13.

### 4.2.1.2 Declaring Big Objects

The following header is used for declaring a big structured object:

**struct sBCELL** Structure  
 Include this structure as a first component of your data structure if it will be either small or big.

A typical example is the declaration of dynamically sized arrays:

```
typedef struct sARRAY {
    struct sBCELL header;
    /* ... data ... */
} * ARRAY;
```

The region following the header will hold the arrays data. Most probably this will become a deep structured object, since the array data consists of objects again.

There are two macros associated with this kind of cells which allow you to access the data as an array of objects and the size of this array.

**OBJ\* data\_big** (OBJ *obj*) Macro  
 For a structured object declared as big, return a pointer to the first object in the data area.

**WORD size\_big** (OBJ *obj*) Macro  
 For a structured object declared as big, return the size in objects of the data area.

### 4.2.2 Allocating Structured Objects

The deepness or flatness of structured objects is determinated at allocation time. This information is stored in the cells header, such that deallocation can be performed without knowledge about deepness or flatness. This is crucial to realize parametric resp. polymorphic objects.

Six macros for allocation of structured objects are predefined. The size argument must always be calculated with one of the size macros given below.

**void alloc\_small** (WORD *size*, OBJ *res*) Macro  
**void alloc\_small\_flat** (WORD *size*, OBJ *res*) Macro  
**void alloc\_small\_byte\_flat** (WORD *size*, OBJ *res*) Macro  
 Allocate a deep, flat, or byte-flat small structured object of *size*, and store its reference in *res*.

**void alloc\_big** (WORD *size*, OBJ *res*) Macro  
**void alloc\_big\_flat** (WORD *size*, OBJ *res*) Macro  
**void alloc\_big\_byte\_flat** (WORD *size*, OBJ *res*) Macro  
 Allocate a deep, flat, or byte-flat big structured object of *size*, and store its reference in *res*.

There are several macros to determinate the required size of a structured object:

**WORD sizeof\_small** (*type*) Macro

Use this macro to determinate the required size of a small structured object of fixed size. The argument is the layout type; e.g. `sizeof_small(struct sREAL)`.

**WORD sizeof\_big** (*type*) Macro

Use this macro to determinate the required size of a big structured object of fixed size. The argument is the layout type.

**WORD size\_data** (*size\_t datasize*) Macro

Use this macro to determinate the size of a small or big structured object with dynamic calculated size. *datasize* determinates the size of the data area, excluding the (big or small) header, e.g. `size_data(n * sizeof(OBJ))` where *n* is the number of objects which shall fit in the data area.

## 4.3 Garbage Collection and Reference Counting

The approach of reference counting to garbage collection is rather simple. The cell of a structured object contains a counter holding the number of all the references pointing to it, the so-called *reference counter*. If this counter drops to zero, the cell becomes isolated and can be collected as free memory.

To make the scheme work, everytime a new reference is created, the reference counter must be incremented, and everytime one is deleted, it must be decremented. Creating a reference is done by one of the allocation functions — in which case the reference counter is correctly initialized — or by *copying* an existing reference. Deleting a reference happens finally when the life-time of an automatic variable holding a reference expires.

The reference counting scheme relies on the fact that there are no cyclic dependencies between structured objects. The compiler ensures this per definition; the handcoder himself is responsible for cyclic dependencies in the handcoded structures.

### 4.3.1 Reference Counting of Arbitrary Objects

The runtime system provides several macros to copy and delete references. The most general macros are used in such situations in which it is not clear whether an object is structured or primitive:

**void copy\_some** (OBJ *obj*, int *cnt*) Macro

Increment the reference counter of *obj* by *cnt*, if *obj* is structured.

**void free\_some** (OBJ *obj*, int *cnt*) Macro

Decrement the reference counter of *obj* by *cnt*, if *obj* is structured. If the reference counter drops to zero, collect the cell for the free memory pool.

### 4.3.2 Reference Counting of Structured Objects

If its is known that an object is structured, the more specific macros for reference counting should be used.



**void copy\_structured** (OBJ *obj*, int *cnt*) Macro  
 Increment the reference counter of structured object *obj* by *cnt*.

**void free\_structured** (OBJ *obj*, int *cnt*) Macro  
 Decrement the reference counter of structured object *obj* by *cnt*. If the reference count drops to zero, collect the cell for the free memory pool.

### 4.3.3 Selective Update

*Selective Updating* is an important source of optimization used in the OPAL compiler. The following macros may be used to realize it on the level of handcoding:

**int excl\_structured** (OBJ *obj*, int *cnt*) Macro  
 Test if the reference counter of structured *obj* is exactly *cnt*.

**void decr\_structured** (OBJ *obj*, int *cnt*) Macro  
 Decrement the reference counter of structured *obj* by *cnt*. No test is performed if the counter drops to zero. This function should be used only if it is known that the counter is greater than *cnt*.

A typical example how to use these macros for selective update is an update function on handcoded arrays:

```
OBJ _AArray_Aupd(A,i,x){
  OBJ result;
  if (excl_structured(A,1)){
    result=A;
  } else {
    result=duplicate_array(A); decr_structured(A,1);
  }
  FREE(data_big(result)[unpack_word(i)],1);
  data_big(result)[unpack_word(i)] = x;
  return result;
}
```

Here, `duplicate_array` is an auxiliary function which duplicates an array.

### 4.3.4 Borrowing References

In order to allow for maximal sources of selective updating, the number of references to some cell should be minimized. The *borrowing* technique may be used for this purpose. It also allows deep objects to become freed as flat objects; this is significantly faster, since the runtime system has not to perform subfrees on the components any more.

Borrowing is based on the following macros:

**void dispose\_structured** (OBJ *obj*) Macro  
 Dispose the cell of a structured object. This should be done only if it is known that the cell is isolated.

**void dispose\_structured\_flat** (OBJ *obj*) Macro

Dispose the cell of a structured object and force it to be flat. This should be done only if it is known that the cell is isolated and that all structured objects it contains have been borrowed.

**OBJ NIL** Constant

A generic primitive object used for borrowing.

Borrowing is not very often performed in handcoded structures, since it is mainly of interest for recursive data types. Those data types should be implemented in OPAL; here the compiler does most probably a better job. Anyway, to illustrate borrowing (as it is also performed by the compiler), we define a recursive sequence-like type as follows:

```
typedef struct sSEQ {
    struct sCELL header;
    OBJ ft;
    OBJ rt;
} * SEQ;
```

A function which selects both the **ft** and the **rt** element from a sequence **s** now typically contains the following code:

```
...
{ OBJ ft = s->ft, rt = s->rt;
  if (excl_structured((OBJ)s,1)) {
    /* borrow ft and rt from s,
       and treat s as flat. */
    dispose_structured_flat((OBJ)s);
  } else {
    copy_some(ft,1); copy_some(rt,1);
    decr_structured((OBJ)s,1);
  }
}
```

A function which selects only the **rt** element has two possibilities for borrowing. The first one is to explicitly clear the **rt** component using the constant **NIL**:

```
...
{ OBJ rt = s->rt;
  if (excl_structured((OBJ)s,1)) {
    s->rt = NIL;
    dispose_structured((OBJ)s);
  } else {
    copy_some(rt,1);
    decr_structured((OBJ)s,1);
  }
}
```

This version has the disadvantage that a flat dispose cannot be performed, since the **ft** component of **s** is still bounded.

The second version explicitly frees the **ft** component in order to perform a flat dispose. This version, however, makes only sense in this particular example, since the number of components which must be freed is rather small:

```

...
{ OBJ rt = s->rt;
  if (excl_structured((OBJ)s,1)) {
    free_some(s->ft,1);
    dispose_structured_flat((OBJ)s);
  } else {
    copy_some(rt,1);
    decr_structured((OBJ)s,1);
  }
}

```

### 4.3.5 Simulating Storage Classes

In certain circumstances it is useful to fix the storage classes the compiler assigns to hand-coded data types in order to give it more exact information about the data type.

Assume, for instance, the data types of natural numbers (**nat**) and arrays (**array**) — both of which are hand-coded in the OPAL standard library. Natural numbers certainly are primitive objects, no reference handling, therefore, has to be done on natural numbers. In order to give the compiler the information not to perform reference counting (e.g. in the functions it generates automatically) a simulated storage class can be given by

```
DATA nat == somePrimitive
```

Arrays always are deep structured objects. To give the compiler this information a simulated storage class can be given by

```

SORT dummy
DATA Array == someStructured(dummy1: dummy, dummy2: dummy)

```

Note that it is necessary to give at least two components in the declaration, since the compiler otherwise eliminates the constructor, and that sort **dummy** should not be implemented such that the compiler does not know its storage class. To simulate a flat structured object, you would have for example sort **bool** instead of **dummy**.

In the functions generated automatically the compiler now can avoid the test whether an array is primitive or structured (e.g. before copying a reference).

Note that this feature will be provided in a much cleaner way by specialized compiler *pragmas* in the near future. (See *The Programming Language OPAL*)

## 4.4 Conventions for Declaring Data Types and Related Functions

We recommend to use the following conventions to declare hand-coded data types which are exported accross structure boundaries. They are also used for the hand-coded structures of the OPAL standard library, and allow yourself and others to deduce the existence and semantics of functions more easily.

With each introduced primitive object type *TYPE*, supply a type definition and corresponding *pack\_type* und *unpack\_type* functions. For example, the hand-coded interface part of the structure **Nat** from the OPAL library starts as follows:

```

/* hand-coded interface part of Nat */

/* representation */

typedef WORD NAT;

#define pack_nat(x)    pack_word(x)
#define unpack_nat(x)  unpack_word(x)

/* macro based implementations */
...

```

With each introduced structured object type *TYPE*, supply a structure definition *sTYPE* and a type definition *TYPE* which is a pointer to the structure, together with the following macros:

|  |       |
|--|-------|
| <b>void alloc_type</b> (OBJ <i>res</i> )   | Macro |
| Allocate an (uninitialized) object and store it in <i>res</i> .                                    |       |
| <b>void make_type</b> (... <i>data</i> ..., OBJ <i>res</i> )                                       | Macro |
| If useful for this type: allocate an object, initialize it with data, and store it in <i>res</i> . |       |
| <b>void copy_type</b> (OBJ <i>obj</i> , int <i>cnt</i> )   | Macro |
| <b>void free_type</b> (OBJ <i>obj</i> , int <i>cnt</i> )   | Macro |
| Copy and free macros for type <i>TYPE</i> .  |       |
| <b>int excl_type</b> (OBJ <i>obj</i> , int <i>cnt</i> )  | Macro |
| <b>void decr_type</b> (OBJ <i>obj</i> , int <i>cnt</i> )   | Macro |
| <b>void dispose_type</b> (OBJ <i>obj</i> )   | Macro |
| <b>void dispose_type_flat</b> (OBJ <i>obj</i> )  | Macro |
| If useful for this type: macros for selective update and borrowing.                                |       |
| <b>OBJ* data_type</b> (OBJ <i>type</i> )   | Macro |
| <b>OBJ* size_type</b> (OBJ <i>type</i> )   | Macro |
| Return data area and data size of object.  |       |

For example, the handcoded interface part of the structure *Real* from the OPAL library starts as follows:

```

/* hand-coded interface part of Real */

#include <math.h>

/* representation */

typedef struct sREAL {
    struct sCELL header;
    double value;
} * REAL;

```

```

#define alloc_real(r)      alloc_small_flat(sizeof_flat(struct sREAL),r)
#define make_real(v,r)     {alloc_real(r); ((REAL)(r))->value = v; }
#define copy_real(o,n)     copy_structured(o,n)
#define free_real(o,n)     free_structured(o,n)
#define excl_real(o,n)     excl_structured(o,n)
#define decr_real(o,n)     decr_structured(o,n)
#define dispose_real(o,n)  dispose_structured(o,n)

    /* macro based implementations */
...

```

## 4.5 Allocating Auxiliary Memory

The runtime system of OCS uses its own memory allocation methods and free memory pool. Memory allocated by the runtime system will never be released such that it can be reused by the C library function `malloc` and its derivatives. Hence, if you require auxiliary memory for your handcoded implementations, you should use the following functions for allocating and freeing it:

**void\* alloc\_aux (size\_t size)** Function

Like the C standard library function `malloc`, but allocates from the OCS runtime system memory pool. This function always return properly aligned pointers for use of `pack_pointer` and `unpack_pointer` (see Section 4.1.3 [Pointer Primitives], page 11).

**void free\_aux (void\* mem)** Function

Like the C standard library function `free`, but frees memory formerly allocated by `alloc_aux`.

## 5 Functions

One of the characteristics of the compilation scheme used by OCS is to use no special function calling protocol, but to imitate as closely as possible the normal C calling conventions. This should enable the C compiler to optimize function calls w.r.t. the specific target architecture. For handcoding of OPAL structures it has the pretty side-effect that handcoded or compiler generated OPAL functions are almost plain C functions.

However, one has to be aware that functions are first-order citizens in OPAL, but not in C. This problem is coped with as follows: for each global OPAL function exists one C data object, the so-called *closure object* of type OBJ. This is the representation of a function as a first-order citizen, which may be passed around as a parameter and stored in other data objects. It is subject of reference counting as are other data objects. See Section 5.4 [Closure Objects], page 22.

Furthermore, for each global OPAL function exists a *direct call entry*. This is actually a plain C function which can only be used in full applications (that is, applications which supply all arguments of a function according to its rank, see Section 5.1 [Ranks], page 20). The direct call entry is what you actually code in C. The closure objects are automatically created by the derived C implementation part of a structure. See Section 5.2 [Direct Entries], page 21.

To support the efficient realization of handcoded basic data types there is third kind of C object associated with each OPAL function: the *macro expansion entry*. You can supply macro definitions of functions in the handcoded interface part of a structure. If you don't give macro entries, default ones are supplied by the derived interface part. See Section 5.3 [Macro Entries], page 21.

The different kinds of calling entries only exist for 'real' functions with rank greater than zero. The value of OPAL Functions of rank zero are evaluated at initialization time of a structure, and stored in a global variable. See Section 5.5 [Constants], page 23.

### 5.1 Ranks and eta-Enrichment

The *rank* of a function is the number of arguments supplied in its definition. For example, the definition `DEF @(o)(x,y) == o(x,y)` has rank 3 and the definition `DEF @(o) == o` has rank 1, although both definitions (nearly) represent the same semantics (if *o* represents the same function in both cases).

Unfortunately, the rank of a function is a private property of the implementation of a structure. If inter-structure optimizations are enabled, this property is propagated and exploited for optimizations (creating recompilation dependencies between structure implementations), but there is currently no way to access it in handcoded structures.

To allow the prediction of function ranks the OPAL compiler performs what we call *eta-enrichment* of *handcoded* structures. This means that additional arguments are appended to the right and left hand sides of function definitions according to the arity of the functionality of the function. If `FUN @ : (s ** s -> s) -> s ** s -> s`, for example, the definition `DEF @(o) == o` will be actually translated to `DEF @(o)(x,y) == o(x,y)`. Hence you can predict

the rank from the arity of the functionality which is a non-private property visible in the signature part of a structure.<sup>1</sup>

## 5.2 Direct Call Entry

The direct call entry of a function is named *\_basename* (for function basenames see Section 3.2 [Basenames], page 8.) At least this function has to be supplied for each function which is to be handcoded.

The direct call entry takes as many arguments of type OBJ as the functions rank is. If the result of a function is not a tuple, it returns a single OBJ. For example, the following direct entry implements the successor function on natural numbers:

```
OBJ _ANat_Asucc(OBJ n) {
    return pack_nat(unpack_nat(n) + 1);
}
```

If the result of a function is a tuple, the direct entry returns one of the predefined tuple structure types *TUP<sub>n</sub>* from the runtime system:

**TUP<sub>n</sub>** Type

Used to represent tuple results of functions. Genereally defined as:

```
typedef struct { OBJ c1, c2, .... , cn; } TUPn;
```

For example, a possible implementation of the function *divmod* on natural numbers is:

```
TUP2 _ANat_Adivmod(OBJ n, OBJ m) {
    TUP2 res; NAT cn = unpack_nat(n); NAT cm = unpack_nat(m);
    res.c1 = cn / cm; res.c2 = cn % cm;
    return res;
}
```

Note that when the direct function entry is called, it *owns* the references of structured object parameters. If the function uses a parameter *n* times by passing it to other functions or returning it, it has to perform *n-1* copies of the reference; if *n* is zero, it has to perform one free on the reference. This naturally generalizes to local objects created in the course of a computation.

## 5.3 Macro Expansion Entry

The macro expansion entry of a function is named *basename* (for function basenames see Section 3.2 [Basenames], page 8.) It allows for inline expansion of short function definitions. Inline expansion is crucial for basic data types like natural numbers; handcoded structures making use of this feature, however, may not port to long-term future generations of OCS.

The macro entry takes as many arguments as a the functions rank, and additional *l*-values used to store the result(s) of the function evaluation. The sucessor function on naturals may be coded as a macro entry as follows:

```
#define ANat_Asucc(x,r) {r=pack_nat(unpack_nat(x) + 1);}
```

For functions with tuple results, consecutive result parameters are supplied:

---

<sup>1</sup> This is not yet realized in Version 2.1a; you have to do it by hand

```
#define ANat_Adivmod(n,m,r1,r2){ \
    NAT cn = unpack_nat(m), cm = unpack_nat(m); \
    r1 = pack_nat(cn / cm); r2 = pack_nat(cn % cm);\
}
```

You usually place the macro entry of a function in the handcoded interface part of a structure. Be aware of including headers of other handcoded structures, if you refer to definitions supplied by them. For example, if the function `divmod` is implemented in a separate structure `DivMod` based on the structure `Nat`, the header has to include `Nat.h`:

```
/* handcoded interface part of DivMod */
#include "Nat.h"
#define ANat_Adivmod(n,m,r1,r2){ \
    NAT cn = unpack_nat(m), cm = unpack_nat(m); \
    r1 = pack_nat(cn / cm); r2 = pack_nat(cn % cm);\
}
```

If you have implemented handcoded functions using macros, you still have to supply the direct call entry. This is necessary, since the direct entry is used for constructing the closure object of the function. The usual method in this case is to implement the direct entry using the macro entry:

```
OBJ _ANat_Asucc(OBJ n) {
    OBJ res;
    ANat_Asucc(n,res);
    return res;
}
```

Please note, that the compiler uses the macro entries of imported handcoded structures only if inter-structure optimization is enabled. This is due to the fact that the pragma `/$ handcoded $/` is a property private to an implementation.

## 5.4 Closure Objects

The closure object of a function is named `__basename` (for function basenames see Section 3.2 [Basenames], page 8.) You never construct closure objects by yourself; they are constructed by the runtime system for you.

To evaluate a closure, the literal way is to use the macro `METHOD` from the runtime system to select an evaluation method and then to call this method:

|  |       |
|--|-------|
| <pre><i>function</i> <b>METHOD</b> (OBJ clos, int argc)</pre>  | Macro |
| Returns a pointer to a function which can evaluate <i>clos</i> with <i>argc</i> arguments.   |       |
| The type of function is physically <code>OBJ function (OBJ clos, OBJ arg1, ..., OBJ argn)</code> ; but you must establish this type by a cast. |       |

Evaluating a closure object with 2 arguments looks as follows:

```
(* (OBJ (*) (OBJ,OBJ,OBJ)) METHOD(clos,2)) (clos,arg1,arg2)
```

To free you from writing down such boring type casts, the runtime systems supplies macros for evaluating closures with upto 8 arguments:



**OBJ EVAL<sub>n</sub>** (OBJ *clos*, OBJ *arg1*, . . . , OBJ *argn*) Macro  
 Selects the appropriate method and evaluates the closure, where *n* must be in the range from 1 to 8.

This section is still to be completed. [ Missing: Evaluating closures with tuple results. ]

### 5.4.1 Contents of Closures

Generally, a closure consists of a pointer to the direct call entry of a function, a pointer to a table of methods used to evaluate the closure, and any arguments which have been used to construct the function the closure represents. You do not have to understand the details but a short explanation will be useful later on.

If  $\text{FUN } o : (b \rightarrow c) ** (a \rightarrow b) \rightarrow a \rightarrow c$  is the usual function composition, then the application  $f \circ g$  denotes a new function constructed from applying *o* to *f* and *g*, represented by a closure which holds a pointer to the direct call entry of *o* and the arguments *f* and *g*.

To obtain the particular evaluation method necessary to evaluate a given closure, the table of evaluation methods carried by the closure is indexed by the number of arguments the closure shall be evaluated with. There are basically three kinds of methods for evaluating a closure:

1. If there are still missing arguments according to the rank of the direct call entry stored in the closure, a new closure is constructed holding the stored and the supplied arguments with the same direct entry call and an appropriately updated table of evaluation methods.
2. If the stored arguments together with the supplied arguments match the rank of the stored direct call entry, this is called with all arguments.
3. If the stored and the supplied arguments extend the rank, the direct entry is called with as many arguments as required and the result (which must be a closure for type correct programs) is evaluated with the remaining arguments as its arguments.

### 5.4.2 Reference Counting of Closures

Closures are in principal structured objects and thus need reference counting. Two functions are used for reference counting on closures:

**void copy\_closure** (OBJ *clos*, int *cnt*) Macro  
 Increment the reference counter of closure *clos* by *cnt*.

**void free\_closure** (OBJ *clos*, int *cnt*) Macro  
 Decrement the reference counter of closure *clos* by *cnt*. If the reference count drops to zero, collect the closure for the free memory pool.

## 5.5 Constants

The value of OPAL functions of rank zero are calculated at structure initialization time and stored in a global variable. This variable is named `__basename` (for function basenames see Section 3.2 [Basenames], page 8).

Please note that this name coincides with the closure object name of a function. Actually, a function of rank zero might very well represent a higher order object and the variable then holds a closure object. Since eta-enrichment is performed on handcoded structures, this practically does not occur in the world of handcoding, since the rank is always identical with the arity (see Section 5.1 [Ranks], page 20).

A handcoded structure must provide a special static function named `init_const_CIde`, where *CIde* is the transliterated identifier of the structure (see Section 3.1 [Transliteration], page 8). This function initializes all constants of the structure. For example, the structure `Nat` may contain initialization code as follows:

```
static init_const_ANat() {
    __ANat_A0 = pack_nat(0);
    __ANat_Amax = pack_nat(max_word);
}
```

Mutual dependent constant systems have to be initialized in a proper order. Since the OPAL compiler has no idea of dependencies of handcoded constants from OPAL coded constants of the same structure, it assumes that there are no such dependencies. Hence, the handcoded constants are always initialized *before* the other constants and thus cannot refer to them.<sup>2</sup>

---

<sup>2</sup> The constant initialization scheme might be subject of changes in near future.

## 6 Builtin data types

The data types of boolean values and textual denotations are builtin into the runtime system.

The operations on builtin types are declared in *BOOL.sign* and *DENOTATION.sign*. However, these are only pseudo structures, and the associated declarations on the C level are incorporated into the runtime system headers, which are automatically included in the handcoded parts.

In this chapter the operations predefined on the two builtin types are described shortly.

### 6.1 Booleans

The following functions operating on boolean values are available:

|  |       |
|--|-------|
| <b>OBJ pack_bool</b> (int <i>value</i> )                                     | Macro |
| Packs an integer to a boolean value; 0 becomes false and anything else true. |       |
| <b>OBJ pack_clean_bool</b> (int <i>flag</i> )                                | Macro |
| Used for packing if <i>flag</i> equals exactly to 0 or 1                     |       |
| <b>int unpack_bool</b> (OBJ <i>obj</i> )                                     | Macro |
| Unpacks a boolean value  |       |

The function **true**, **false**, **~**, **and**, **or** etc., as declared in *BOOL.sign*, are available as canonical implementations, that is, the (closure) variables, direct and macro entries are supplied as usually. But note that the origin is not *BOOL* but *BUILTIN*; hence the basename of **true** is e.g. *ABUILTIN\_Atrue* (see Section 3.2 [Basenames], page 8).

### 6.2 Denotations

Since denotations are structured objects, memory management has to be addressed in this case as described in Section 4.4 [Data Type Conventions], page 17.

|   |      |
|---|------|
| <b>DENOTATION, sDENOTATION</b>  | Type |
| Data type definition (for the definition of <i>sBCELL</i> see Section 4.2.1.2 [Big Objects], page 13) |      |

```
typedef struct sDENOTATION {
    struct sBCELL big;
    OBJ leng;                /* packed length */
    /* ... data ... */      /* data */
} * DENOTATION;
```

|  |          |
|--|----------|
| <b>OBJ alloc_denotation</b> (int <i>leng</i> )                                       | Function |
| Allocates an uninitialized denotation of given length.                               |          |
| <b>OBJ make_denotation</b> (char* <i>cstr</i> )                                      | Function |
| Allocates a denotation and initializes it with the given C zero-terminated C string. |          |

|  |          |
|--|----------|
| <b>int</b> <b>get_denotation</b> ( <i>OBJ den</i> , <i>char* buf</i> , <i>int bufsize</i> )  | Function |
| Copies the data of a given denotation in a buffer of a given length, with a trailing zero terminator, and frees the denotation. This function returns 1 if the denotations and the terminator fits into the buffer. It returns 0 and only copies the truncated string data if not.   |          |
| <b>char*</b> <b>charbuf</b>  | Variable |
| <b>int</b> <b>CHARBUFSIZE</b>  | Macro    |
| In order to facilitate the use of <i>get_denotation</i> and other string processing functions, this function predefines a character buffer of size <b>CHARBUFSIZE</b> which equals — in the current release — to 1024. You should not call any other function of the runtime system when using this buffer, since they might globber it. |          |
| <b>char*</b> <b>data_denotation</b> ( <i>OBJ den</i> )   | Macro    |
| Returns a char pointer to the data of a denotation. Note that denotations are not zero-terminated.   |          |
| <b>WORD</b> <b>leng_denotation</b> ( <i>OBJ den</i> )  | Macro    |
| Returns the length of the denotation.  |          |
| <b>void</b> <b>free_denotation</b> ( <i>OBJ den</i> , <i>int cnt</i> )   | Macro    |
| <b>void</b> <b>copy_denotation</b> ( <i>OBJ den</i> , <i>int cnt</i> )   | Macro    |
| <b>void</b> <b>excl_denotation</b> ( <i>OBJ den</i> , <i>int cnt</i> )   | Macro    |
| <b>void</b> <b>decr_denotation</b> ( <i>OBJ den</i> , <i>int cnt</i> )   | Macro    |
| Usual RC operations on denotations. <sup>1</sup>   |          |

As for boolean numbers, for the functions declared in **DENOTATION.sign** canonical C declarations exist; however, the origin to be used for the basename is **BUILTIN** rather than **DENOTATION**.

---

<sup>1</sup> Bug: the function **dispose\_denotation** is missing in ocs 2.1a.

## 7 The OPAL Library

The OPAL standard library provides a rich set of structures. Some of them are handcoded, and for all structures a handcoding interface exists such that they may be accessed from other handcoded structures.

In the following sections the a handcoding interface for basic types (**Nat**, **Int**, **Real**), an aggregate types (**String**), and structures used to access system services (**Com**, **Process**) are presented.

### 7.1 Basic Types

*Nat, Int*     The handcoding interface part `Nat.hc.h` only defines packing routines but no memory management routines for natural numbers, since naturals are primitive. The handcoding interface part `Int.hc.h` is very similar – except of course that all operations work on signed values.

|                                     |       |
|-------------------------------------|-------|
| <b>NAT</b>                          | Type  |
| OBJ <b>pack_nat</b> (NAT)           | Macro |
| Packs to an OPAL <code>nat</code> . |       |
| NAT <b>unpack_nat</b> (OBJ)         | Macro |
| Unpacks an OPAL <code>nat</code> .  |       |

The OPAL structure `Nat.sign` defines natural numbers as a free type with constructors `0` and `succ` and several arithmetic and boolean operations on them. These operations can be accessed in handcoded structures by their transliterated names, e.g. `_ANat_Asucc`, as usual (see Section 3.1 [Transliteration], page 8).

*Real*     In contrast to the preceding two basic types, the type of real numbers is structured, so memory management routines have to be defined for them (see Section 4.4 [Data Type Conventions], page 17). The mathematical functions defined in the signature `Real.sign` can be accessed via their transliterated names as usual.

Real numbers are defined as

|  |      |
|--|------|
| <b>struct sREAL</b>  | Type |
| <pre>typedef struct sREAL {     struct sCELL header;     double value; } * REAL;</pre> |      |

The following memory management macros are also defined in `Real.hc.h`:

|   |       |
|---|-------|
| <b>void alloc_real</b> (OBJ <i>res</i> )              | Macro |
| Allocates a real and stores it in lvalue <i>res</i> . |       |
| <b>void make_real</b> (double, OBJ <i>res</i> )       | Macro |
| Allocates and initializes a real.                     |       |

|   |       |
|---|-------|
| <code>void free_real (OBJ,int)</code>       | Macro |
| <code>void copy_real (OBJ,int)</code>       | Macro |
| <code>void excl_real (OBJ,int)</code>       | Macro |
| <code>void decr_real (OBJ,int)</code>       | Macro |
| <code>void decr_real (OBJ,int)</code>       | Macro |
| Usual RC operations on reals <sup>1</sup> . |       |

## 7.2 Aggregate Types

*String* Acting for the other numerous aggregate types of the OPAL standard library, the handcoding interface of the structure implementing strings (`string.h`, `string.hc.h`) is described here in some more detail.

In contrast to the primitive types described in see Section 7.1 [Basic Types], page 27, memory management is essential to use aggregate types in handcoded structures.

While the functions operating on strings which are declared in the signature file `String.sign` can be accessed by their transliterated names (see Section 3.1 [Transliteration], page 8, the handcoding interface `String.hc.h` offers the following routines to perform memory management:

|   |          |
|---|----------|
| <code>int get_string (OBJ str, char* buf, int bufsize)</code>   | Function |
| Copies the OPAL string <code>o</code> into the buffer <code>s</code> of size <code>n</code> and frees it.<br>This function returns '1' if the string completely fits in the buffer, '0' if not (and truncates the string to the maximal length). The string in the buffer is zero-terminated. |          |
| <code>OBJ make_string (char *)</code>   | Macro    |
| This macro creates an OPAL string from a C string.  |          |
| <code>int is_empty_string(OBJ)</code>   | Macro    |
| This macro tests for the empty string.  |          |
| <code>void unpack_chunk_string (OBJ, int, OBJ, OBJ)</code>  | Macro    |
| Unpack components of a string, a chunk.   |          |
| <code>OBJ addr_rest_string(OBJ)</code>  | Macro    |
| Get address of the rest field of a chunk.   |          |

*Array* Arrays are an important data structure posing interesting problems with respect to functional programming languages. The handcoding interface `Array.hc.h` declares the following routines (confer to Section 4.4 [Data Type Conventions], page 17).

---

<sup>1</sup> Bug: `dispose_real` missing in ocs-2.1a

## 7.3 Operating System

OPAL uses a monadic approach to input and output. The IO monad is called *command* and is of sort `com'Com`. Technically, commands are data types which are passed to the run time system which interpretes them, thus performing side-effects.

This section describes commands in more detail, shows how commands can be constructed on the handcoding level, and exemplifies their use by presenting how commands are used in the OPAL library to access the services of the process abstraction of UNIX.

### 7.3.1 I/O-Handling Using Commands

*Monads* are a new and powerful approach to incorporate i/o-handling in functional programming languages.

In OPAL monads are implemented by *commands*. Commands are terms of a particular free type (defined in structure `Com`). All functions exhibiting side-effects (possibly by calling other functions with side-effects) are thus characterized by a return type of `com[data]`.

The free type is defined in `Com.sign` as

```
TYPE com ==
  yield      (ans: ans)
  exit       (value: nat)
  call       (proc: void -> ans)           -- embedding side-effect call
  followedBy (com: com, cont: ans -> com)  -- composing commands
  sync       (proc: void -> ans)           -- embedding thread sync call
  resume     (cont: ans -> com)           -- resume after sync
```

The handcoding of functions with side-effects thus proceeds by the following two phase scheme:

1. Declare a function on the OPAL level with return type `com[data]` and implement it constructing a term of type `com` using a handcoded function performing the side-effect.
2. Implement the handcoded function itself. In order to adjust the evaluation of functional expressions the handcoded function has to be implemented as a particular higher-order function (depending on the function on the OPAL level). If the handcoded function should take `n` parameters `p1`, ..., `pn`, and return a value of type `data`, its functionality has to be `FUN handcodedFunction : p1 ... pn -> void -> ans[data]`

As an example, consider the implementation of the function `close`, which closes a UNIX file.

`File.sign`

```
FUN close : file -> com[void]
```

The OPAL signature declares the function “close”.

`File.impl`

```
DEF close(f) == call(xclose(f))
```

```
FUN xclose : file -> void -> ans[void]
```

The OPAL implementation defines the function by constructing a term of type `com` which — in this case — only calls the handcoded function `xclose`. Furthermore, it declares the function `xclose`. Note the extra `void` parameter.

**File.hc.c**

The handcoded function only calls the corresponding UNIX function and ensures the proper translation of parameters and return values:

```
extern OBJ _AFile_Axclose(OBJ file,OBJ unit) {
    FILE *f = unpack_file(file);
    if (fclose(f) != EOF){
        return_okay_nil;
    } else {
        return_unix_failure(errno);
    }
}
```

### 7.3.2 Constructing Commands in Handcoded Structures

Section Section 7.3.1 [Commands], page 29 described the concept of commands and showed how to manage interfacing between OPAL and handcoded functions. In the handcoded interface, however, additional functions and macros are used to construct commands which are defined in the handcoding interface `Com.hc.c`. These are listed in the following table.

**ans\_okay\_nil**

A constant object representing a success answer without further data.

**return\_okay\_nil**

A macro returning a success answer without data.

**return\_okay(data)**

A macro returning a success answer with data.

**return\_fail(failure)**

A macro returning a given failure answer. The failure conditions returned from UNIX functions are predefined in `UnixFailures.h`, users can define additional failure message using the following function.

**OBJ declare\_failure\_answer(char \* message)**

This function can be used to construct new failure answers which can be returned using `return_fail`.

For an example using these functions/macros see Section 7.3.1 [Commands], page 29.

### 7.3.3 Example: Accessing the UNIX Process Abstraction

In order to give a more complete example of interfacing between OPAL and handcoded structures using commands, this section describes the process abstraction implementing that of UNIX in the OPAL standard library.

**Process.sign**

This interface declares OPAL equivalence for UNIX (heavyweight) process and pipe abstraction. It declares the functions `self`, `self?`, `fork`, `execve`, `kill`, `wait`, `popen`, and `pclose`





## 8 Embedding OPAL Programs into C Programs

In order to call OPAL programs from C programs, the C-routine implementing the OPAL program has to be called as exemplified by the function `main` used defined in `_ostart.c`.

```
main(int argc, char** argv, char** environ){
    OBJ ans;

    start_argc = argc; start_argv = argv;
    start_env = environ;

    init_ABUILTIN();
    init();

    COPY(command,1);
    return MAIN(command);
}
```

For this function to work three global variables must be defined:

|                      |   |
|----------------------|---|
| <code>command</code> | The C function implementing the main OPAL function, i.e. representing the top-level command |
| <code>init</code>    | The init entry of the structure containing <code>command</code>                             |
| <code>include</code> | The foreign interface of the structure containing <code>command</code>                      |

## 9 Upgrading Handcoded Structures

Prior versions of the OPAL-compiler — up to version 2.0h — used a slightly different handcoding scheme. This section describes the changes and how to upgrade to the handcoding scheme defined in this document.

This chapter is still incomplete.

### 9.1 Interfacing to OPAL

Keeping the handcoded part independent from the OPAL implementations was a major design decision taken in the OPAL compiler up to version 2.0h. Due to this design decision handcoding had to be done on three (instead of two) levels. In between the proper OPAL and handcoded structures the handcoder had to insert the so-called “RTS”-level.

The definitions of that level have been shifted to the OPAL implementation part of a handcoded structure with compiler version 2.1a which also entailed a simplification of the argument and return value passing scheme.

Consider interfacing to the startup routine of a graphical user interface. Besides an OPAL function `initXmInterface`

```
FUN initXmInterface    : com[xmDialog]

DEF initXmInterface ==
  abs(call(mapAns(abs: RTSXmDialog->xmDialog,
               initXmInterface'RTSXmDialog)))
```

you had to declare an additional function on the RTS-level and handle the proper translation of argument and return value types.

```
FUN initXmInterface : RTSUnit[RTSXmDialog] -> RTSans[RTSXmDialog]
```

With the handcoding scheme introduced in version 2.1a, this declaration is completely done on the OPAL level by

```
FUN initXmInterface    : com[xmDialog]

DEF initXmInterface ==  call(xinitXmInterface)
FUN xinitXmInterface: void -> ans[xmDialog]
```

### 9.2 Transliteration of Identifiers

In the compiler version 2.1a minor changes to the lexical rules of identifiers (confer to *The Programming Language OPAL*) have taken place. The most important one is that underscores (‘\_’) are allowed as part of an identifier in version 2.1a.

Due to this fact the transliteration scheme had to be changed. In the following the old transliteration scheme is given

- Each generated name starts with the structure name followed by an underscore (‘\_’).
- Next, the character sequence ‘v\_’ is appended, if the generated name denotes a closure variable.

- The name of the function is then appended as it appears in the OPAL source. The transliteration of special symbols is done as shown in Section Section 3.1 [Transliteration], page 8.
- Finally, an additional number is appended in case of overloaded functions.

The most important difference thus is that transliterated groups now have to be preceded by an ‘A’ (alphanumeric) or ‘S’ (special).

### 9.3 Macro Naming

This section is still to be written.

### 9.4 Object Declaration

This section is still to be written.

### 9.5 Memory Allocation

This section is still to be written.

# Index

|  |    |
|--|----|
| -  |    |
| <code>_ALIGN_POINTERS</code>                       | 11 |
| <code>_ostart.c</code>                             | 32 |
| <b>A</b>   |    |
| accessing plain structures                         | 7  |
| <code>addr_rest_string(OBJ)</code>                 | 28 |
| aggregate type                                     | 28 |
| <code>alloc_aux</code>                             | 19 |
| <code>alloc_big</code>                             | 13 |
| <code>alloc_big_byte_flat</code>                   | 13 |
| <code>alloc_big_flat</code>                        | 13 |
| <code>alloc_denotation</code>                      | 25 |
| <code>alloc_real</code>                            | 27 |
| <code>alloc_small</code>                           | 13 |
| <code>alloc_small_byte_flat</code>                 | 13 |
| <code>alloc_small_flat</code>                      | 13 |
| <code>alloc_type</code>                            | 18 |
| <code>ans_okay_nil</code>                          | 30 |
| arithmetic functions, unchecked                    | 27 |
| array  | 28 |
| auxiliary memory                                   | 19 |
| <b>B</b>   |    |
| basename   | 8  |
| basic data type                                    | 27 |
| big object   | 13 |
| <code>bits_per_word</code>                         | 11 |
| booleans, <code>BOOL.sign</code>                   | 25 |
| borrowing  | 15 |
| builtin data type                                  | 25 |
| <b>C</b>   |    |
| <code>charbuf</code>                               | 26 |
| <code>CHARBUFSIZE</code>                           | 26 |
| closure object                                     | 22 |
| <code>com</code>                                   | 29 |
| command  | 29 |
| constant   | 23 |
| <code>copy_closure</code>                          | 23 |
| <code>copy_denotation</code>                       | 26 |
| <code>copy_real</code>                             | 28 |
| <code>copy_some</code>                             | 14 |
| <code>copy_structured</code>                       | 15 |
| <code>copy_type</code>                             | 18 |
| <b>D</b>   |    |
| data type, aggregate                               | 28 |
| data type, basic                                   | 27 |
| data type, builtin                                 | 25 |
| data type, conventions                             | 17 |
| <code>data_big</code>                              | 13 |
| <code>data_denotation</code>                       | 26 |
| <code>data_type</code>                             | 18 |
| <code>declare_failure_answer</code>                | 30 |
| <code>decr_denotation</code>                       | 26 |
| <code>decr_real</code>                             | 28 |
| <code>decr_structured</code>                       | 15 |
| <code>decr_type</code>                             | 18 |
| <code>DENOTATION</code> , <code>sDENOTATION</code> | 25 |
| denotations, <code>DENOTATION.sign</code>          | 25 |
| direct call entry                                  | 21 |
| <code>dispose_structured</code>                    | 15 |
| <code>dispose_structured_flat</code>               | 16 |
| <code>dispose_type</code>                          | 18 |
| <code>dispose_type_flat</code>                     | 18 |
| <b>E</b>   |    |
| eta enrichment                                     | 20 |
| <code>EVALn</code>                                 | 23 |
| <code>excl_denotation</code>                       | 26 |
| <code>excl_real</code>                             | 28 |
| <code>excl_structured</code>                       | 15 |
| <code>excl_type</code>                             | 18 |
| <b>F</b>   |    |
| <code>FOREIGNSTRUCTS</code>                        | 6  |
| <code>free_aux</code>                              | 19 |
| <code>free_closure</code>                          | 23 |
| <code>free_denotation</code>                       | 26 |
| <code>free_real</code>                             | 28 |
| <code>free_some</code>                             | 14 |
| <code>free_structured</code>                       | 15 |
| <code>free_type</code>                             | 18 |
| <b>G</b>   |    |
| garbage collection                                 | 14 |
| <code>get_denotation</code>                        | 26 |
| <code>get_string</code>                            | 28 |
| <b>H</b>   |    |
| handcoded structure                                | 3  |

|  |   |
|--|---|
| handcoded structure, derived parts .....       | 4 |
| handcoded structure, implementation part ..... | 3 |
| handcoded structure, maintenance .....         | 6 |
| handcoded structure, signature part .....      | 3 |
| handcoded structure, source files .....        | 3 |

## I

|                                      |        |
|--------------------------------------|--------|
| i/o-handling .....                   | 29     |
| identifier, transliteration of ..... | 8, 33  |
| INT .....                            | 27     |
| integer .....                        | 11, 27 |
| is_empty_string(OBJ) .....           | 28     |
| is_primitive .....                   | 10     |
| is_structured .....                  | 12     |

## L

|                              |    |
|------------------------------|----|
| leng_denotation .....        | 26 |
| library, OPAL standard ..... | 27 |

## M

|  |    |
|--|----|
| macro entry .....                            | 21 |
| main .....                                   | 32 |
| make_denotation .....                        | 25 |
| make_real .....                              | 27 |
| make_string .....                            | 28 |
| make_type .....                              | 18 |
| max_sword .....                              | 11 |
| max_word .....                               | 11 |
| memory management, garbage collection .....  | 14 |
| memory management, reference counting .....  | 14 |
| memory management, selective update .....    | 15 |
| memory management, structured data object .. | 13 |
| memory, auxiliary .....                      | 19 |
| METHOD .....                                 | 22 |
| min_sword .....                              | 11 |
| monads .....                                 | 29 |

## N

|   |    |
|---|----|
| struct sREAL .....                      | 27 |
| natural numbers .....                   | 10 |
| natural numbers, integral numbers ..... | 27 |
| NIL .....                               | 16 |
| NORMSTRUCTS .....                       | 6  |

## O

|                             |    |
|-----------------------------|----|
| OPAL standard library ..... | 27 |
|-----------------------------|----|

## P

|                 |    |
|-----------------|----|
| pack_bool ..... | 25 |
|-----------------|----|

|  |    |
|--|----|
| pack_clean_bool .....                        | 25 |
| pack_int .....                               | 27 |
| pack_nat .....                               | 27 |
| pack_pointer .....                           | 11 |
| pack_sword .....                             | 11 |
| pack_word .....                              | 10 |
| pointer .....                                | 11 |
| primitive data object .....                  | 10 |
| primitive data object, integer .....         | 11 |
| primitive data object, natural numbers ..... | 10 |
| primitive data object, pointer .....         | 11 |
| Process.hc.h, Process.hc.c .....             | 30 |
| Process.sign, Process.impl .....             | 30 |

## R

|   |    |
|---|----|
| rank .....                              | 20 |
| real .....                              | 27 |
| reference counting .....                | 14 |
| return_fail .....                       | 30 |
| return_okay .....                       | 30 |
| return_okay_nil .....                   | 30 |
| RTS, intermediate interface level ..... | 33 |

## S

|  |    |
|--|----|
| sBCELL .....                                 | 13 |
| sCELL .....                                  | 12 |
| selective update .....                       | 15 |
| side effect .....                            | 29 |
| simulating storage class .....               | 17 |
| size_big .....                               | 13 |
| size_data .....                              | 14 |
| size_type .....                              | 18 |
| sizeof_big .....                             | 14 |
| sizeof_small .....                           | 14 |
| small object .....                           | 12 |
| sREAL .....                                  | 27 |
| string .....                                 | 28 |
| struct sBCELL .....                          | 13 |
| struct sCELL .....                           | 12 |
| struct sREAL .....                           | 27 |
| structure, handcoded .....                   | 3  |
| structured data object .....                 | 12 |
| structured data object, big .....            | 13 |
| structured data object, memory management .. | 13 |
| structured data object, small .....          | 12 |
| SWORD .....                                  | 11 |

## T

|                                      |       |
|--------------------------------------|-------|
| target language .....                | 1     |
| transliteration of identifiers ..... | 8, 33 |

`TUP $n$`  ..... 21  
type ..... 25, 27, 28  
type discipline ..... 2

## U

unchecked arithmetic functions ..... 27  
`unpack_bool` ..... 25  
`unpack_chunk_string` ..... 28

`unpack_int` ..... 27  
`unpack_nat` ..... 27  
`unpack_pointer` ..... 11  
`unpack_sword` ..... 11  
`unpack_word` ..... 10

## W

`WORD` ..... 10

## Short Contents

|   |   |    |
|---|---|----|
| 1 | Introduction . . . . .                            | 1  |
| 2 | Handcoded Structures . . . . .                    | 3  |
| 3 | Naming Conventions . . . . .                      | 8  |
| 4 | Data Objects and Memory Management . . . . .      | 10 |
| 5 | Functions . . . . .                               | 20 |
| 6 | Builtin data types . . . . .                      | 25 |
| 7 | The OPAL Library . . . . .                        | 27 |
| 8 | Embedding OPAL Programs into C Programs . . . . . | 32 |
| 9 | Upgrading Handcoded Structures . . . . .          | 33 |
|   | Index . . . . .                                   | 35 |



# Table of Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction .....</b>   | <b>1</b>  |
| 1.1      | What's the use of Handcoding .....                                  | 1         |
| 1.2      | Target and Foreign Language .....                                   | 1         |
| 1.3      | Warning .....   | 1         |
| <b>2</b> | <b>Handcoded Structures .....</b>                                   | <b>3</b>  |
| 2.1      | Source Parts of Handcoded Structures .....                          | 3         |
| 2.2      | A Look and Feel Example .....                                       | 4         |
| 2.3      | Derived Parts of Handcoded Structures .....                         | 4         |
| 2.4      | Maintaining Handcoded Structures .....                              | 6         |
| 2.5      | Accessing Plain Structures from Handcoded Structures .....          | 7         |
| <b>3</b> | <b>Naming Conventions .....</b>                                     | <b>8</b>  |
| 3.1      | Transliteration of OPAL Identifiers .....                           | 8         |
| 3.2      | Basenames of Functions .....  | 8         |
| <b>4</b> | <b>Data Objects and Memory Management ...</b>                       | <b>10</b> |
| 4.1      | Primitive Objects .....   | 10        |
| 4.1.1    | Unsigned Integers as Primitive Objects .....                        | 10        |
| 4.1.2    | Signed Integers as Primitive Objects .....                          | 11        |
| 4.1.3    | Pointers as Primitive Objects .....                                 | 11        |
| 4.2      | Structured Objects .....  | 12        |
| 4.2.1    | Declaring Structured Data .....                                     | 12        |
| 4.2.1.1  | Declaring Small Objects .....                                       | 12        |
| 4.2.1.2  | Declaring Big Objects .....   | 13        |
| 4.2.2    | Allocating Structured Objects .....                                 | 13        |
| 4.3      | Garbage Collection and Reference Counting .....                     | 14        |
| 4.3.1    | Reference Counting of Arbitrary Objects .....                       | 14        |
| 4.3.2    | Reference Counting of Structured Objects .....                      | 14        |
| 4.3.3    | Selective Update .....  | 15        |
| 4.3.4    | Borrowing References .....  | 15        |
| 4.3.5    | Simulating Storage Classes .....                                    | 17        |
| 4.4      | Conventions for Declaring Data Types and Related Functions<br>..... | 17        |
| 4.5      | Allocating Auxiliary Memory .....                                   | 19        |
| <b>5</b> | <b>Functions .....</b>  | <b>20</b> |
| 5.1      | Ranks and eta-Enrichment .....                                      | 20        |
| 5.2      | Direct Call Entry .....   | 21        |
| 5.3      | Macro Expansion Entry .....   | 21        |
| 5.4      | Closure Objects .....   | 22        |

