

FFS Users Guide and Reference

Greg Eisenhauer

September 16, 2014 – FFS Version 1.2

1 Introduction

FFS (Fast Flexible Serialization) is a system for efficiently marshaling data for communication or storage in a heterogeneous computing environment. This manual serves as both an introduction to using FFS and as a reference manual for the API.

FFS is more complex than the type systems built into many middleware environments because it was designed to operate efficiently in situations where *a priori* knowledge shared between the sender (writer) and receiver (reader) is limited. In particular, FFS :

- tries to minimize copying, so as not to interfere with delivering communication bandwidth and latency near what the underlying network,
- supports system evolution through flexible matching between incoming and expected data (limited *a priori* knowledge),
- allows the receipt and manipulation of data that is unknown before runtime (zero *a priori* knowledge),
- has the ability to enact run-time specified functions to data types that are potentially unknown until run-time, with an efficiency near native code.

2 FFS Basics

The basic approach of the FFS library is relatively simple. FFS supports both files and message-based transport, but at its core FFS is record-oriented. From a data-description point of view, writers (or encoders) of data must provide a description of the names, types, sizes and positions of the fields in the records they are writing. Readers/receivers must provide similar information for the records that they are interested in reading.

FFS uses this ‘format’ information to establishing a correspondence between the fields in the incoming records and the fields in the local data structure into which the information is to be placed. No translation is done on the writer’s end. On the reader’s end, the format of the incoming record is compared with the format that the program expects. If there are discrepancies, either in the data layout or contents, or in the underlying machine representations, the FFS read routine performs the appropriate translations.

2.1 Describing Data/Message Types

The FFS routines support records consisting of fields of basic atomic data types, including: “integer”, “unsigned integer”, “float”, and “string”. Note that *field type* here is separate from *field size* so that both the native C types `int` and `long` are “integer” types. Similarly, C’s `double` and `float` are both declared as “float”

In order to support record-oriented structure reading and writing, FFS must know the exact layout of the structure, including name, type, starting offset and size of each field. This information is conveyed to

FFS using an `FMFieldList`¹ with an entry for each field. Because the size and offset of fields varies from architecture to architecture (and even from compiler to compiler), it is good practice to use the `sizeof()` operator to calculate field size. FFS also provides a macro, `FMOffset()` that uses compile-time tricks to calculate the offset of a field with a structure. Below is an example structure for a record and the corresponding field list:

```
typedef struct _first_rec {
    int      i;
    long     j;
    double   d;
    char     c;
} first_rec, *first_rec_ptr;

static FMField field_list[] = {
    {"i", "integer", sizeof(int), FMOffset(first_rec_ptr, i)},
    {"j", "integer", sizeof(long), FMOffset(first_rec_ptr, j)},
    {"d", "float",   sizeof(double), FMOffset(first_rec_ptr, d)},
    {"c", "integer", sizeof(char), FMOffset(first_rec_ptr, c)},
    {NULL, NULL, 0, 0},
};
```

The order of fields in the field list is not important. It is not even necessary to specify every field in a structure with an entry in the field list. Unspecified fields at the end of the structure may not be written to the IO file. Unspecified fields at the beginning or in the middle of a structure will be written to the IO file, but no information about them will be available.²

2.2 Simple Programs

FFS supports both file-oriented and online (network-based) data interchange. The various uses of FFS share a common underlying infrastructure with respect to type manipulation. We'll first demonstrate the file interface. FFS uses several opaque types as handles to internal data structures:

FMContext - `FMContext`'s are repositories of structure and type information in FFS. Sometimes an `FMContext` is used on its own and sometimes in concert with other handles.

FFSFile - a handle to a file used for reading or writing FFS data. `FFSFile` have internal `FMContext`'s.

FMFormat - an `FMFormat` represents a handle to a registered structure type.

The code below represents a simple FFS program that writes a record to a file. We use the struct declaration and `FMFieldList` given in the example above (though we left out the initialization of the written data structure for simplicity).

¹The 'FM' prefix is associated with Format Manager functionality within FFS.

²*Nota Bene:* This behavior results because FFS treats base data records as a contiguous memory block for efficiency. The compiler may leave "holes" in the data block to preserve the required alignment for some data types. Because FFS puts the data on the wire as a contiguous block, the values in these "holes" are put on the wire as well. Fields in the record that are not described to FFS via the field list are simply treated as larger "holes". Note that relying on the behavior of FFS with respect to values in these "holes" is erroneous and can have unpredictable results. For example, "hole values" may appear to be transmitted along with the data in an exchange between homogeneous systems, yet not in other cases. That "hole values" are read by FFS may also confuse valgrind and other tools who detect the operation as a read of uninitialized memory.

```

int main(int argc, char **argv)
{
    FFSFile file = open_FFSfile("test_output", "w");
    FMContext fmc = FMContext_of_file(file);
    FMFormat rec_format;
    first_rec rec1;

    rec_format = FMregister_simple_format(fmc, "first rec", field_list, sizeof(first_rec));
    write_FFSfile(file, rec_format, &rec1);
    close_FFSfile(file);
}

```

This program creates a file named `test_output`. You can use the program `FFSdump`, part of the FFS package to print this binary file as verbose text. `FMregister_simple_format()` is one of the most basic calls in FFS, used to provide FFS with the detailed field layout of a structure. FMformats are registered with particular FMContexts, this one extracted from the FFSFile. In addition to the FMFieldList, structures/types must have a name and you must provide its size. The `FMregister_simple_format()` call can only be used if the structure contains only basic atomic types, or arrays of or pointers to basic atomic types. Structures which contain other structures must be registered with `FMregister_data_format()` as described in Section 3.1.

The code below is a simple FFS program that reads the file written above. Once again we use the struct declaration and FMFieldList given earlier and while we've left the print statements out of this code for simplicity, you can find them in the actual programs in the `ffs/doc/examples` directory in the FFS source code distribution.

```

int
main(int argc, char** argv);
{
    FFSFile file = open_FFSfile("test_output", "r");
    FFSTypeHandle first_rec_handle;
    FFSContext context = FFSContext_of_file(file);
    first_rec rec1;

    first_rec_handle = FFSset_simple_target(context, "first rec", field_list, sizeof(first_rec));
    FFSread(file, &rec1);
    close_FFSfile(file);
}

```

The code above demonstrates the basics, albeit with several simplifying assumptions. Structurally it is similar to the writing program. `FFSset_simple_target()` call provides FFS with a description of a particular structure/message type that this program can handle and is the read-side analogue of the `FMregister_simple_format()` call. The `FFSTypeHandle` is roughly equivalent to an FMFormat, except that it is associated with incoming messages/records rather than outgoing ones. FFSContext uses here is similar to the FMContext, except that it also stores the associations between the “original” source format (the ‘wire’ format) and the “native”, or target format. The `FFSread()` calls grabs the next record in the file and puts it in the `rec1` variable. In this case, we are relying upon the *a priori* knowledge that there is one record in the file and that its format *compatible* is compatible with the native *first_rec*.

Compatibility The notion of format *compatibility* is fairly important in FFS. The general idea is that the target format specifies a set of required fields and the physical layout that the reading program requires them in. FFS matches source and target fields by name and any source format which contains the required target fields is potentially compatible. In the example above, the reader and writer are using the same C-level structure declaration. If the programs were executed on different architectures, their binary representations (and hence the details of the source and target formats) might differ in byte order, type size and layout,

but FFS would compensate for all of these differences and deliver the data to the receiver in his native representation. FFS will also ignore (or correct for) differences in field order and even the presence of additional fields not required by the reader. These techniques, and others which are described later, help FFS enable communication even when multiple versions of application components in complex distributed systems.

3 More Complexity in Types

The example structure in the previous section consisted solely of fields of simple atomic data types. FFS actually supports much more complexity than those simple structures. In particular, FFS supports nested structures, null-terminated strings, both statically- and variable-sized arrays, pointers to simple and complex objects and tree- and graph-based structures.

3.1 Nested structures

Multiple structures may make up a complete datatype. FFS requires that the transitive closure of all referenced structures be declared to FFS together as a unified type.

For example, the structure `particle_struct` declared as in Figure 1 could be described to FFS through the following field lists and a new element, the `FMStructDescRec` : field lists:

```
typedef struct R3vector_struct {
    double x, y, z;
} R3vector;

typedef struct particle_struct {
    R3vector loc;
    R3vector deriv1;
    R3vector deriv2;
} particle;
```

Figure 1: A nested record format

```
static FMField R3field_list[] = {
    {"x", "float", sizeof(double), FMOffset(R3vector*, x)},
    {"y", "float", sizeof(double), FMOffset(R3vector*, y)},
    {"z", "float", sizeof(double), FMOffset(R3vector*, z)},
    {NULL, NULL, 0, 0},
};

static FMField particle_field_list[] = {
    {"loc", "R3vector", sizeof(R3vector), FMOffset(particle*, loc)},
    {"deriv1", "R3vector", sizeof(R3vector), FMOffset(particle*, deriv1)},
    {"deriv2", "R3vector", sizeof(R3vector), FMOffset(particle*, deriv2)},
    {NULL, NULL, 0, 0},
};

FMStructDescRec particle_format_list[] = {
    {"particle", particle_field_list, sizeof(particle), NULL},
    {"R3vector", R3field_list, sizeof(R3vector), NULL},
    {NULL, NULL, 0, NULL}};
```

There are a couple of things to note here. First, the `FMStructDescRec` essentially contains, for each of the structures, the elements passed to describe a single structure in `FMregister_simple_format()`. (Ignore the fourth, `NULL`, element in each entry for the moment.) Second, the first entry in each `FMStructDescRec` is the structure name and it is that name that is used as a *field type* in the field lists of containing structures. Lastly, the first element in the `FMStructDescRec` list must be top-level structure. The remaining structure entries can appear in any order, but they together they must constitute the transitive closure of all structures referenced.

Once the appropriate `FMStructDescRec` list has been assembled, the structure can be substituted into the

reader and writer programs above by substituting `FMregister_data_format()` and `FFSset_fixed_target()` calls for `FMregister_simple_format()` and `FFSset_simple_target()`, respectively.

```
extern FMFormat
FMregister_data_format(FMContext context, FMStructDescList struct_list);

extern FFSTypeHandle
FFSset_fixed_target(FFSContext c, FMStructDescList struct_list);
```

3.2 Array types

In previous examples, the *field type* entry in field lists is always the name of an atomic type or of a structure listed in the `FMStructDescList`, however FFS allows more complex field types to be represented. Fields of which are arrays of other elements are specified by adding an array size declaration to the the *field type* entry. For example, `"float[5]"` and `"unsigned integer[10]"` both declare fields which are statically sized arrays. Multiple size declarations can be supplied in order to declare multiply dimensioned arrays, such as `"float[2][12]"`. These field types match the C declarations in Figure 2. Note that when declaring array fields to FFS, the *field size* element in the `FMField` list must be the *element* size, not the total array size. When reading a record containing array fields, the dimensionality of each field must match that which was used when the record was written, though the size of the elements may differ.

```
struct {
    float      a[5];
    unsigned int b[10];
    float      c[2][12];
}
```

Figure 2: Simple Array fields

In addition to fixed array sizes, FFS supports dynamically sized arrays, where the array size is controlled by data values at run-time. In this case, the size in the array type specification must be the string name of an integer field in the record. The value of that integer field gives the array size. The actual data type in the record should be a pointer to the element type. Figure 3 gives an example of a dynamic array declaration. Dynamic arrays may also be multiply dimensioned, including the use of a mix of static and dynamic sizes.

```
typedef struct _dyn_rec {
    char      *string;
    long      count;
    double    *double_array;
} dyn_rec, *dyn_rec_ptr;

FMField dyn_field_list[] = {
    {"string field", "string", sizeof(char *),
     FMOffset(dyn_rec_ptr, string)},
    {"count", "integer", sizeof(long),
     FMOffset(dyn_rec_ptr, count)},
    {"double_array", "float[count]", sizeof(double),
     FMOffset(dyn_rec_ptr, double_array)},
    { NULL, NULL, 0, 0}
};
```

Figure 3: A dynamic array record format

3.3 Pointer-based Types

Unlike its predecessor P BIO, FFS supports pointer-based declarations in field types. These are declared by using a `*` in the field type string. For example, the field type `"*integer"` declares a pointer to a single integer and `"*integer[5]"` declares a pointer to a fixed size array of 5 integers. These basic features can be combined, using parenthesis to control association. For example, while `"*integer[5]"` is a pointer to 5 integers, `"(*integer)[5]"` is a fixed size array of 5 pointers to integers. Note that the use of a dynamic array bound (*as in* `"float[count]"`) implies a pointer-based type. In fact, `"float[count]"` and `"*float[count]"` specify identical types to FFS.³ Note: For all the pointer-based types, the *field size* element in the `FMField` list must be the size of item pointed-to, or the element size if the final item is an array.

Of course, in addition to pointers to atomic data types and arrays of those types, FFS allows pointers

³There is no way to specify a non-pointer-based dynamically sized type to FFS. Only statically-sized arrays may have storage in-line with the other fields in a structure.

```

typedef struct node {
    int node_num;
    struct node *link1;
    struct node *link2;
} *node_ptr;

FMField node_field_list[] =
{
    {"node_num", "integer", sizeof(int), FMOffset(node_ptr, node_num)},
    {"link1", "*node", sizeof(struct node), FMOffset(node_ptr, link1)},
    {"link2", "*node", sizeof(struct node), FMOffset(node_ptr, link2)},
    {(char *) 0, (char *) 0, 0, 0}
};

FMStructDescRec node_format_list [] = {
    {"node", node_field_list, sizeof(struct node), NULL},
    {NULL, NULL, 0, NULL}
};

```

Figure 4: An example of a recursively-defined type

to structures, including recursively-defined structures. This allows FFS to represent and transport pointer-based structures such as linked lists, graphs and trees. Figure 4 shows the FFS declaration of a structure that implements a binary tree. When working with recursively-defined types, FFS will write/encode the transitive closure of all data blocks pointed to in the structure presented. It keeps a visit table to ensure each block is only processed once, even if its address appears in multiple pointers. When the message is read/decoded, pointers which originally targeted the same block will still point to the same block in the decoded message.

4 Using FFS Over Networks or Other Transport

The simple example programs given above use the FFSfile interface to FFS, but FFS was designed to operate in message-oriented mode with fully-typed messages being exchanged in a heterogeneous environment. The FFS APIs used in this environment use the terms *encoding* and *decoding*, but marshaling and unmarshaling would be just as appropriate. Because network-based applications can be a bit tricky to setup and require multiple programs to run simultaneously, we're going to demonstrate these APIs with simple programs that write the encoded messages to files. The principles are the same and the reader should be able to work the the programs more easily.

4.1 Simple Encode Program

The non-file-oriented FFS programs use the same basic types as the file-based programs: FMContext, FFSTypeHandle, FMFormat and FFSContext.

The code below represents a simple FFS program that encodes a record of the type used in the first examples and writes the encoded data to a file. (Once again, we've left out the initialization of the written data structure for simplicity.)

```

int main(int argc, char **argv)
{
    FMContext fmc = create_FMContext();
    FFSBuffer buf = create_FFSBuffer();
    FMFormat rec_format;
    first_rec rec1;
    char *output;
    int fd, output_size;

    rec_format = register_simple_format(fmc, "first rec", field_list, sizeof(first_rec));
    output = FFSencode(buf, rec_format, &rec1, &output_size);

    /* write the encoded data */
    fd = open("bin_output", O_WRONLY|O_CREAT);
    write(fd, output, output_size);
    close(fd);
}

```

There are several differences between this encode sample and the `write_FFSfile()` example above. Instead of extracting the `FMContext` value that was created with the `FFSfile`, this example creates an `FMContext` value directly. The `FMContext` data structure is the principal repository of structure and type information in FFS. Section 8 describes `FMContexts` and how they operate in more detail.

This sample code also uses an `FFSBuffer`. `FFSBuffers` are simply a convenient mechanism for reusing memory from one `FFSencode` to the next. Each `FFSBuffer` wraps a malloc'd data block, tracking its allocated size and current usage. `FFSencode` places the encoded data in the malloc'd block, realloc'ing it if necessary depending upon the size of the encoded block. The data within the block remains valid until the `FFSBuffer` is used in another FFS call.

The principal routine here is the `FFSencode()`. Like the `write_FFSfile()`, `FFSencode` takes an `FMFormat` and a pointer to the data to be encoded. The `FFSBuffer` fills in for the `FFSfile` pointer as the destination. However, it also takes an integer pointer that is filled in with the actual size of the encoded data. The return value is a pointer to beginning of the block of encoded data (which may not correspond to the beginning of the `FFSBuffer`'s data block.) Generally the length of the data block is necessary for decoding the data, so it is usually transmitted, explicitly or implicitly, along with the data itself. Here, since there is a single message stored in the file, we are implicitly using the file length as a means of communicating the block length to the receiver.

```

int main(int argc, char **argv)
{
    FMContext fmc = create_FMContext();
    FFSBuffer buf = create_FFSBuffer();
    FMFormat rec_format;
    first_rec rec1;
    char *output;
    int fd, output_size;

    rec_format = register_simple_format(fmc, "first rec", field_list, sizeof(first_rec));
    output = FFSencode(buf, rec_format, &rec1, &output_size);

    /* write the encoded data */
    fd = open("bin_output", O_WRONLY|O_CREAT);
    write(fd, output, output_size);
    close(fd);
}

```

4.2 Simple Decode Program

The code below is a simple FFS program that reads the data written into the file above.

```
int main()      /* receiving program */
{
    FFSTypeHandle first_rec_handle;
    FFSContext context = create_FFSContext();
    first_rec rec1;
    int fd, size;
    char encoded_buffer[2048]; /* big enough for this example */

    fd = open("enc_file", O_RDONLY, 0777);
    size = read(fd, encoded_buffer, sizeof(encoded_buffer));

    first_rec_handle = FFSset_simple_target(context, "first rec", field_list, sizeof(first_rec));
    FFSdecode(context, encoded_buffer, (void*)&rec1);
}
```

Just as the encoding program created an FMContext rather than using one associated with an FFSFile, this decoding program creates its own FFSContext. It also uses `FFSset_simple_target()` to tell FFS the details of the structure it wants the data converted to. Then it uses `FFSdecode()` to extract the data from the encoded block and fill in the `first_rec` value.

5 Using FFS With Many Message Types

One of the simplifying assumptions in both the file and message-based examples above is that there is only one type of message being exchanged, so the receiver always knows what data type is expected next. In many practical scenarios this may not be true.

On the sending/encoding side, handling multiple message types is straightforward. Each type must be registered with `FMregister_simple_format()` or `FMregister_data_format()` to get its `FMFormat` value, then that value is used to write or encode data. Type registration may happen all at once when the program starts or the file is opened, or it may happen later, after some other data has been written or encoded. The only requirement is that the registration for a data type happens before it can be written or encoded.

On the receiving side, things are simple in principle, but involve a bit more code. When the receiving program can understand multiple incoming message structures, each should be set as a possible target using `FFSset_simple_target()` or `FFSset_fixed_target()`, reserving the return values. The FFS call `FFS_target_from_encode()` examines an incoming record and determines which of the registered targets it is compatible with (or most compatible with). The corresponding FFSFile call is `FFSnext_type_handle()`. The profiles of these calls are below, followed by an example FFSfile program that reads multiple data types from a file.

```
extern FFSTypeHandle FFSnext_type_handle(FFSFile ffsfile);
extern FFSTypeHandle FFS_target_from_encode(FFSContext c, char *data);
```

The multi-format read program below assumes the visibility of the `first_rec` and `field_list` declarations for earlier in this document. The code is similar to the prior reading program, except that it uses `FFSnext_type_handle()` to identify each incoming record.

```

typedef struct _second_rec {
    int      *items;
    long     count;
    char *   name;
} second_rec, *second_rec_ptr;

static FMField field_list2[] = {
    {"items", "integer[count]", sizeof(int), FMOffset(second_rec_ptr, items)},
    {"count", "integer", sizeof(long), FMOffset(second_rec_ptr, count)},
    {"name", "string", sizeof(char*), FMOffset(second_rec_ptr, name)},
    {NULL, NULL, 0, 0},
};

int
main(int argc, char** argv)
{
    FFSFile iofile = open_FFSfile("test_output", "r");
    FFSTypeHandle first_rec_handle, second_rec_handle, next_handle;
    FFSContext context = FFSContext_of_file(iofile);

    first_rec_handle = FFSset_simple_target(context, "first format", field_list, sizeof(first_rec));
    second_rec_handle = FFSset_simple_target(context, "second format", field_list2, sizeof(second_rec));
    while ((next_handle = FFSnext_type_handle(iofile)) != NULL) {
        if (next_handle == first_rec_handle) {
            first_rec rec1;
            FFSread(iofile, &rec1);
            printf("Read first_rec : i=%d, j=%ld, d=%g, j=%c\n", rec1.i, rec1.j, rec1.d, rec1.c);
        } else if (next_handle == second_rec_handle) {
            second_rec rec2;
            int i;
            FFSread(iofile, &rec2);
            printf("Read items= ");
            for (i=0; i<rec2.count; i++) printf("%d ", rec2.items[i]);
            printf(", count = %d, name = %s\n", rec2.count, rec2.name);
        } else {
            printf("Unknown record type file\n");
        }
    }
    close_FFSfile(iofile);
}

```

6 Memory Handling

There are two aspects of memory handling that this section will concern itself with: alternative FFS interfaces that minimize data copies and understanding what FFS is doing with data that it stores. Optimizing memory handling, typically to minimize copies, is a performance concern in FFS, particularly for large records. There are fewer user-visible aspects of this to be concerned with in the FFSFile interface, but the FFSContext interface provides different variations of its interfaces that might be more optimal in different circumstances.

6.1 Encode memory handling

For example, the `FFSencode()` call in the writing program of Section 4.1 must copy every byte of the data to be encoded in order to create a contiguous encoded block. This contiguous data block is actually stored in temporary memory associated with the `FFSContext` (therefore the value returned by `FFSencode()` should not be passed to `free()`). The data is only guaranteed to remain valid until the next FFS operation

on that FFSContext.

Encoding to a contiguous block is necessary if you are dealing with a transport or storage mechanism that requires data to be presented as such. However, if the data is going to be presented to a system call like `writew()` that accepts a vector of data buffers, then copying everything is unnecessary. In this circumstance, it's best to use `FFSEncode_vector()` which copies on the minimum amount of data, leaving the rest in place.

```
typedef struct FFSEncodeVec {
    void *iov_base;
    unsigned long iov_len;
} *FFSEncodeVector;

extern FFSEncodeVector
FFSEncode_vector(FFSBuffer b, FMFormat fmformat, void *data);
```

The `struct FFSEncodeVec` structure which results from `FFSEncode_vector()` is identical to `struct iovec` that is used with `writew()` and similar calls. The `FFSEncode_vector()` call also uses a `FFSBuffer` handle. While `FFSEncode()` copies data into FFS-internal temporary memory to be overwritten at the next FFS operation, `FFSBuffers` are a means through which applications can gain some control over how long encoded data remains valid. Any number of `FFSBuffer` handles can be created with `create_FFSBuffer()` and data encoded into one will remain valid until that `FFSBuffer` is used again. When a `FFSBuffer` is no longer needed, it can be free'd with `free_FFSBuffer(FFSBuffer buf)`.

An example program, similar to that of Section 4.1 follows. We use the `second_rec` data type because it has pointer-based elements.

```
int main(int argc, char **argv)
{
    FMContext fmc = create_FMcontext();
    FFSBuffer buf = create_FFSBuffer();
    FMFormat rec_ioformat;
    second_rec rec2;
    FFSEncodeVector outvec;
    int fd, outvec_size;
    char str[32];
    int array[10];
    int j;

    srand48(time());
    sprintf(str, "random string %ld", lrand48()%100);
    rec2.name = &str[0];
    rec2.count = lrand48()%10;
    rec2.items = &array[0];
    for(j=0; j<rec2.count; j++) rec2.items[j] = lrand48()%5+2;
    rec_ioformat = register_simple_format(fmc, "second format", second_field_list, sizeof(second_rec));
    outvec = FFSEncode_vector(buf, rec_ioformat, &rec2);
    for(outvec_size=0; outvec[outvec_size].iov_len!=0; outvec_size++);

    printf("Wrote name=%s, count=%ld, items = [", rec2.name, rec2.count);
    for(j=0; j<rec2.count; j++) printf("%d, ", rec2.items[j]);
    printf("]\n");

    /* write the encoded data */
    fd = open("enc_file", O_WRONLY|O_CREAT|O_TRUNC, 0777);
    writew(fd, outvec, outvec_size);
    close(fd);
}
```

6.2 Decode memory handling

As usual, the situation on the decoding side is a bit more complex than the encoding side. The example code given in Section 4.2 uses the `FFSdecode()` call, which accepts a contiguous message block as input and places the decoded record in user-supplied memory. This explanation is actually something of a simplification. `FFSdecode()` puts the *base structure* component of the decoded record in user-supplied memory. If the record contains any pointer-based elements, such as strings, variable-sized arrays or other pointer values, those elements are held in FFS-controlled temporary memory associated with the `FFScontext`. This means that they will remain valid only for until the next FFS operation (and that they should never be free'd directly).

Applications which require direct control all message memory, for example to preserve the entire message for later use, should use the routines `FFS_est_decode_length()` and `FFSdecode_to_buffer()`. `FFS_est_decode_length()` returns the number of bytes required to contain the entire decoded contents of a message, including strings, variable arrays, etc.⁴ `FFSdecode_to_buffer()` unmarshals the input record and all its associated pointer-based elements into the provided buffer (assumed to be sized appropriately with `FFS_est_decode_length()`). The record is placed at the beginning of the buffer and it is immediately followed by the pointer-based elements (with padding added if as necessary to respect alignment requirements, etc.) Thus the whole record including the pointer-based elements that it contains can be (must be) treated as a unit for memory allocation. (To be explicit - decoded messages do not have each pointer-based element individually malloc'd. Don't pass those elements to `free()`. Bad things will happen if you do.)

The following is a new version of the decoding program of Section 4.2 which decodes the entire incoming message into malloc'd memory.

```
int main()      /* receiving program */
{
    FFSTypeHandle second_rec_handle;
    FFSContext context = create_FFSContext();
    second_rec *rec2;
    int fd, encode_size, decode_size, j;
    char encoded_buffer[2048]; /* hopefully big enough */

    /* "receive" encoded record over a file */
    fd = open("enc_file", O_RDONLY, 0777);
    encode_size = read(fd, encoded_buffer, sizeof(encoded_buffer));

    second_rec_handle = FFSset_simple_target(context, "second format", second_field_list, sizeof(second_rec));
    FFS_target_from_encode(context, encoded_buffer);
    decode_size = FFS_est_decode_length(context, encoded_buffer, encode_size);
    rec2 = malloc(decode_size);
    FFSdecode(context, encoded_buffer, (void*)rec2);
    printf("Read name=%s, count=%ld, items = [", rec2->name, rec2->count);
    for(j=0; j<rec2->count; j++) printf("%d, ", rec2->items[j]);
    printf("]\n");
}
```

6.3 Decoding In Place

The downside to using `FFSdecode_to_buffer()` is that the entire message is copied into the malloc'd buffer. In many circumstances, the encoded message is not needed once it has been decoded and it would be more efficient if the message could be decoded 'in place', without involving an additional buffer. The routine `FFSdecode_in_place()` is designed for this situation, but due to the nature of FFS encode/decode operations 'in place' decoding isn't always possible. The circumstances here depend upon the characteristics

⁴Because returning an exact size value would essentially require traversing the entire encoded structure, the value returned by `FFS_est_decode_length()` is actually a conservative overestimate.

of the sender and the receiver. Because FFS doesn't marshal to a common "network" format, incoming data is mostly laid out as it was on the sending machine, which may differ from where it is to be decoded. For example, a pointer-based structure encoded on a 32-bit architecture might require more memory when decoded on a 64-bit architecture. FFS has a routine, `FFSdecode_in_place_possible()` that returns true if we can decode 'in place' and false otherwise. `FFSdecode_in_place_possible()` takes a `FFSTypeHandle` parameter, but because in place decoding depends upon the characteristics of the source architecture, the `FFSTypeHandle` here must be the handle of the original encoded data, not the final target type. There is an additional call, `FFSTypeHandle_from_encode()` that returns the original, source-side, `FFSTypeHandle` of an encoded buffer. If `FFSdecode_in_place_possible()` return `FALSE`, the message should be decoded as in prior examples. However if it returns true, the incoming message buffer can also serve as the destination. The following code segment demonstrates the typical logic:

```

if (FFSdecode_in_place_possible(FFSTypeHandle_from_encode(context, encoded_buffer))) {
    FFSdecode_in_place(context, encoded_buffer, (void**)&rec2);
} else {
    decode_size = FFS_est_decode_length(context, encoded_buffer, encode_size);
    rec2 = malloc(decode_size);
    FFSdecode(context, encoded_buffer, (void*)rec2);
}

```

7 On Format Compatibility

Format compatibility is an important concept in FFS and is at the heart of controlling and managing the match-up between incoming messages/records and the data formats that the application is prepared to process.

We'll look first at the most straightforward situation, where the receiving/reading application understands some limited set of message types. In this scenario, a typical non-FFS application would marshal all messages to a known, fixed 'wire' format, perhaps with a unique ID or keyword at the beginning to differentiate different incoming messages. Such an approach is usually efficient, but it depends upon an *a priori* agreement between the senders and receivers, and any change in the wire format must occur simultaneously in all communicating parties. FFS differs mainly in that it allows significant more flexibility on the nature of incoming messages, which allows FFS-based applications to function in environments where *a priori* agreement is not possible, or in situations where message formats may evolve over time.

In the FFS situation, there are two components to the equation, the set of data formats that the application expects and understands (the 'target' set), and the format of each incoming data record. As we have seen above, both of these are specified by `FMStructDescRec` lists that describe the data in detail, including field names, types, sizes and offsets. When an incoming record is presented to FFS, the application generally wants to know "which of my target formats does this incoming message correspond to?" The incoming structure may differ in many characteristics from the target structures, so the question boils down to to which target structure the incoming message can be mapped? In determining possible mappings, FFS considers a number of possible factors, including the name of the top-level data structure, the names and types of the subfields, and the details of substructures. Details of the algorithm employed appear in [2] (the FFS algorithm is essentially identical to the P BIO algorithm presented there). However, the basics of matching can be boiled down to some simple guidelines:

- Field matching is done by name and structure. I.E. to be considered a match, fields must have the same name and the same array dimensionality. Similarly fields with atomic types don't match with fields of the same name that are structures.
- The fields in formats that the application expects (the target set) are considered to be *required* fields. That is, for each target format X and incoming format Y, FFS will consider X to be a match for Y if and only if incoming format Y has a superset of the fields of X.

- If an incoming record matches more than one target format, the target format that matches the most fields in the incoming record is chosen.

Given this matching criterion, the semantics of data reception are straightforward. FFS will convert incoming data into the selected target format with these side effects:

- the byte order, floating point format, pointer size and field size of the incoming data will be changed to match the local native format.
- if the matching incoming and target fields are of atomic types that differ in size, the data value will be extended or truncated as required.
- extra fields in the incoming data (fields that do not appear in the target format) are ignored.

These semantics allow easy exchange of data between applications running on very different architectures, but they also allow a certain amount of flexibility between senders and receivers that can be helpful in an evolving distributed system. In particular, as opposed to situations which require *a priori* agreements, FFS clients and servers need not necessarily be updated simultaneously when it becomes useful to change the data included in a message.

For example, consider a request-response server with many clients. Suppose we want to include a new field in the request message, perhaps something that specifies a priority or other SLA information for the request. The first step would be to update the server so that it registers both the old and new (with the extra field) formats and handles each appropriately. With the server then handling both types of requests, the clients could be updated gradually, as time allows.

Now suppose further that there is a need to modify the response message, perhaps to include additional information about the resource usage that the request consumed. As long as the new information is represented simply as new fields in the response message, unmodified clients will silently ignore the new data. Clients that want to access the new information can add those fields to their target formats and have them be delivered.

This basic functionality of ignoring “extra” fields in incoming data and being able to register different handlers for multiple versions of incoming data offer a significant improvement in flexibility over systems that require complete *a priori* agreement. However, FFS has two other features that expand what [2] calls the ‘compatibility space’ of an application. The first of these features is simple. For receive-side target formats, FFS allows the specification of ‘default values’. If a field in the target format has a default value, incoming data without that field is still considered a match and the FFS conversion functionality automatically fills the incoming field with the specified default value. Default values are only allowed for atomic data types and are specified in the FMField list by placing the value in parenthesis after the ‘field name’. E.G.

```
{"priority(7)", "unsigned integer",
    sizeof(int), FMoffset(request_msg_ptr, priority)},
```

While the prior FFS features, ignoring extraneous fields and adding default values are useful, they are still limited. Sometimes one would like to make changes to message types where the new and old type contain basically the same information, but perhaps it is organized differently. Maybe a ‘total cost’ field is broken down into ‘price per item’ and ‘item count’. Maybe multiple fields are grouped into a new substructure. Perhaps an array is converted to a linked list. *Message morphing*, as described in [2] is an approach to extending the compatibility space to include these sorts of changes. While a full description of how to employ message morphing is beyond the scope of this manual, the basic idea is simple. When creating the new message format, the developer includes a small code snippet that can rewrite the new message into its older form. The code snippet is expressed in COD, the FFS data processing language that is described below in Section 10. From the format compatibility point of view, the use of message morphing changes the

format matching procedure above in that it adds a second possible option to what format the incoming data can be. If the matching routines can't find a match in the target set for the incoming data's base format, it will then consider the format into which the morphing code can transform the data and see if there is a match for that format. If there is, all incoming data is automatically transformed into the 'older' format by the morphing code. This allows 'new' servers to interact easily with 'old' clients and vice versa, without resorting to techniques such as protocol negotiation.

8 Working with Formats

In order to achieve the semantics described above, the receiver must have complete knowledge of the nature of the incoming data, including the names and types of the data's fields, their exact layout, size and nature of the underlying bit-representation. Because FFS cannot provide high performance if full structure descriptions and architecture-specific information (byte-order, floating-point format, pointer size) are attached to every message, that *format description* must be separated from the message, itself marshaled and transported separately. Instead, when FFS marshals data for transmission, a 'metadata token', or format ID, is attached to the data as an identifier.

8.1 The Format Server

While there are a variety of ways that FFS might structure its handling of format IDs and format descriptions, there are a couple of key characteristics that must be maintained:

- The format description must contain sufficient information for the receiver to decode the message. Generally this includes the complete structure description and architecture information such as byte-order, floating point format and pointer size.
- The format ID must uniquely identify a format description.
- There must be a mechanism through which a receiver can retrieve the format description given a format ID in an incoming message.

The first requirement ensures that messages can be converted from their wire formats and be interpreted at their destination. The second asserts that there must be no collisions in the format ID space. Note that while there is no requirement in FFS that every format description map to a unique format ID, FFS and EVPath cache and index a variety of information based on format IDs. A mapping that is not 1-to-1 may introduce inefficiencies, but not necessarily incorrectness.

The last requirement comes from anticipated usage of FFS. More accurately, it might be stated that the format description which describes the message must be available to the receiver of that message so that it can be decoded. However, FFS doesn't control the wire transmission, so it can't necessarily ensure that, for example, the format ID/description pair are always transmitted before any messages encoded with that format ID. Even that solution is not guaranteed for multicast transports where a late arriver might miss the initial broadcast, so FFS must provide some kind of a lookup mechanism. Note that there are no particular requirements on the nature of format IDs themselves. They can be as complex as full URLs or as short as necessary for them to be unique, as long as the requirements above are fulfilled.

The format description lookup problem is far from unique, and many well known techniques, such as those that are applied to optimize DNS lookups, can be successfully applied here. Alternatively, one can imagine schemes that exploit the flexibility in the requirements above to embed something like nameserver contact information in the format ID, or where each communicating entity acts as a server for its own formats. We have not attempted to innovate in this space or to solve problems of security and scaling that might arise if FFS were widely deployed, opting instead for a relatively simple solution that still offers relatively efficient lookup.

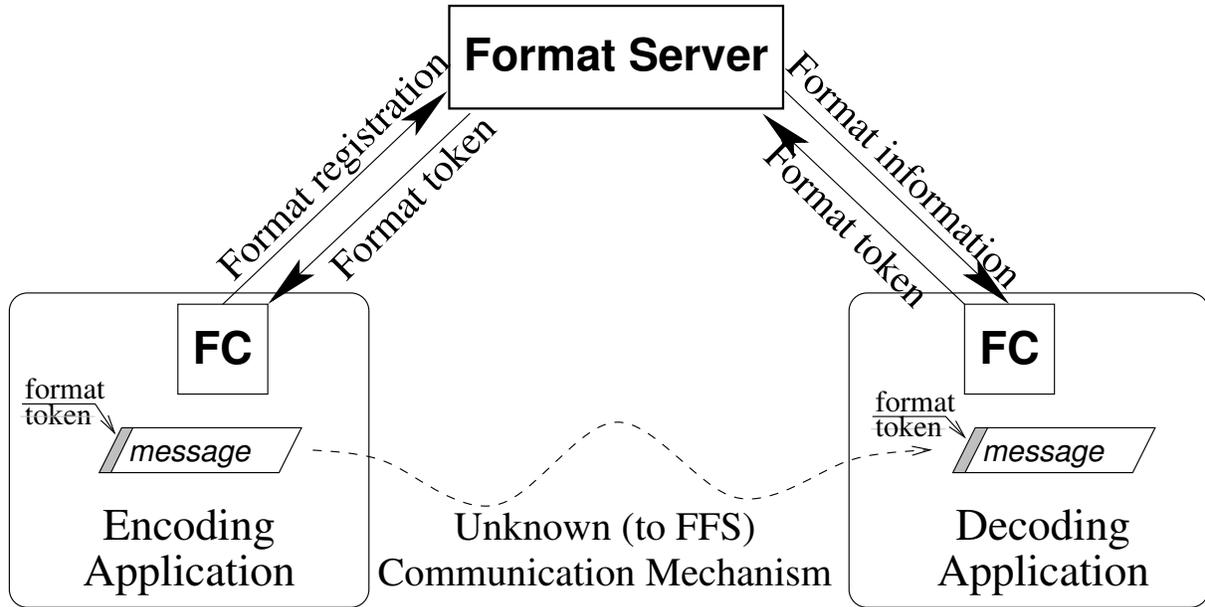


Figure 5: Typical FFS format transfer operation.

Format handling happens at a layer of FFS called Format Manager (FM). The current version of FFS uses 12-byte format IDs which are purely derived from the marshaled format description. The largest component of the format ID is a 64-bit hash of the format description, ensuring that the format ID is unique for each architecture/structure description combination. That the format ID is a pure function of the marshaled format description means that it the format ID can be calculated by the sender without reference to a third party name server. In order to accomplish lookups, Format Manager includes a *format server* that listens for requests on a well-known host and port. The first time a structure description is used in a process to encode a message, the format ID/description pair must be registered with the format server. The location of the format server is configurable at run-time, but all communicating FFS entities must share the same format server. On the receiving side, FM caches format descriptions at several levels for efficiency. In particular, all locally registered format descriptions are available in a per-process cache for queries. In homogeneous transfers, the sender and receiver will both have the same format ID/description pair, so the receiver will find the format description in its cache without having to go to an external server. If not, the format server can be queried to retrieve the appropriate format description.

Typical FFS operation is shown in Figure 5. The format server may run anywhere in the reachable network. When encoding messages applications perform the following basic steps: In format registration to obtain the format token:

1. The local format cache is checked to see if the format being registered is already known.⁵ If the format is known, the format token is taken from the known format.
2. If there is no current connection to the format server, a TCP/IP connection is established.
3. An marshalled version of the format information, including full field lists and native architecture information, is sent to the format server.

⁵I.E. it has already been registered or an identical format is in the cache for other reasons. For formats to be identical all field list entries must be the same, optional information (such as XML representation) must be identical, and the characteristics of the registering architecture (pointer size, byte order, floating point format, etc.) must be the same.

4. The application waits on a `read()` call for the return message from the format server. The return message contains the assigned format token.
5. The format information and format token are entered into the local cache.
6. The connection to the format server is left open for possible reuse. The format server may close it asynchronously.

At this point, the format is available for use on the encoding side. During encoding, the format token is associated with encoded data.

In the decoding application format information retrieved from the format server, typically when the incoming buffer is passed to `get_format_FMcontext()` or similar call. This process follows steps similar to those in the encoding application, specifically:

1. The local format cache is checked to see if the format token in the incoming message is already known.⁶ If the format token is known, the format information associated with that token is used.
2. If there is no current connection to the format server, a TCP/IP connection is established.
3. The format token is sent to the format server.
4. The application waits on a `read()` call for the return message from the format server. The return message either contains the full format information that was registered by the encoding application, or it indicates that the format token is not known. Among other reasons, the latter situation might result from attempting to decode a corrupt message, the encoding and decoding applications using different format servers, or the format server being restarted since the message was encoded.
5. If format information is retrieved from the format server, the information and format token are entered into the local cache.
6. The connection to the format server is left open for possible reuse. The format server may close it asynchronously.

Note that FFS' interactions with the format server may impact application performance. Generally interactions will occur at or near program startup, when message formats are registered, or upon the first appearance of an incoming data item of a particular format, or from a particular architecture.

8.2 Operation without format servers

It is possible for FFS to operate entirely without a format server, if the application itself takes responsibility for ensuring that formats are distributed appropriately. This can be particularly useful in situations where the network is not easily available (such as in the kernel) or where the format server interaction is otherwise undesirable. For example, FFS was used as part of the EVPath infrastructure[1] on tens of thousands of nodes on the Cray "Jaguar" supercomputer at Oak Ridge National Labs. In order not to have tens of thousands of nodes simultaneously hitting the format server to register their (identical) format descriptions, EVPath transported them on the wire to collection sites, where one the first to arrive was sent to the format server for later queries to return.

In order to create an FMContext that is simply a cache with no connection to a format server, one can simply call `create_local_FMcontext()` as a direct substitute for `create_FMcontext()`. That leaves the matter of APIs for extracting the appropriate information from the encoding FMContext and loading it into the receiving.

⁶I.E. a message of that format has been seen before or an identical format is in the cache for other reasons. In this case only the format tokens of known formats are compared.

```

extern char *
get_server_ID_FMformat(FMFormat ioformat, int *id_length);

extern char *
get_server_rep_FMformat(FMFormat ioformat, int *rep_length);

extern FMFormat
load_external_format_FMcontext(FMContext iocontext, char *server_id,
                               int id_size, char *server_rep);

```

The routine `get_server_ID_FMformat()` is used to get the metadata token (format ID) associated with an FMFormat. `get_server_rep_FMformat()` returns the packaged representation of the field names, type and underlying data layout. Each of these is a block of bytes that may contain zero bytes, so they should not be treated as character strings in transmission. The application is free to transport these items to the receiving FMContext with the only restriction being that they must be associated with the receiving context before any data of that format is presented for processing. Once the items arrive at the future receiver of data, they can be associated with the receiving FMContext with the routine `load_external_format_FMcontext()` above.

9 FFS and XML

FFS's XML support had its origins in the observation that FFS and XML both provide similar semantics to the receivers of data. Both formats are self-describing and allow third-parties to interpret and process data about which they have no *a priori* knowledge. Both formats also allow readers to be relatively robust to changes in the data that is sent. In particular, most mechanisms for reading XML (parsers, DOM interfaces, etc.) do not depend upon the ordering of fields in the XML document and are not affected if the XML document contains extra fields or attributes. FFS is similar because of the semantics of the `set_IOConversion()` and `set_conversion_IOcontext()` routines. In particular, the `IOFieldList` supplied in the conversion call specifies the order and layout that the receiver want to see the data in. If the incoming data contains fields in a different order or contains additional fields, those differences are resolved in the conversion process. Fields are reordered and extraneous fields are dropped.

9.1 Just-In-Time Expansion to XML

Because FFS records are completely self-describing, it is possible to “print” them into XML (or virtually any ASCII form) at any point. Two subroutines in FFS specifically support the production of XML. One, `IOencoded_to_XML_string()` operates on data in its encoded form (as produced by `encode_IOcontext_buffer()` for example). The second, `IOunencoded_to_XML_string()` operates on unencoded data. Both routines return a `char*` string value that is the XML representation of the data. The string should be `free`'d when it is no longer needed. The APIs of these routines are:

```

extern char *IOencoded_to_XML_string(IOContext iocontext, void *data);

extern char *IOunencoded_to_XML_string(IOContext iocontext, IOFormat ioformat, void *data);

```

By default, the XML expansion of FFS data is relatively simple and is governed by the format that describes the data. The “format_name” string provided in the original `register_IOcontext_format()` call is used as the name in the surrounding begin/end XML tags for a structure. Within a structure, each fields name is used in begin/end XML tags for that field. Within the field begin/end tags is XML that represents the contents of that field. If the field type is a simple atomic type, the contents are simply printed in an appropriate format. Integer types appear in decimal representations, characters as simple chars, booleans appear as either “true” or “false”, strings appear directly and floating point types are printed using the

“appear as nested XML elements. Arrays appear as repeated XML elements. In arrays of atomic data types, the field name appears as the standard begin/end XML tags where the element contents are the actual array element values with whitespace separators. Fixed-length and variable-length arrays are treated identically.

For example, the record written in the simple encoding example of Section ?? expands to the following XML if `IOencoded_to_XML_string()` is called on the resulting encoded buffer.

```
<dynamic format>
<string field>Hi Mom!</string field>
<icount>5</icount>
<double_array>
0 2.717 5.434 8.151 10.868 </double_array>
</dynamic format>
```

9.2 Customizing the XML Representation

Note that in the default XML representation, all data is presented as an XML element without use of XML attributes. In practice, users may require more control of the manner in which FFS-based records are expanded as XML. In FFS’s model of XML support, additional XML markup information can be associated with `FMFormats` at the time of registration.⁷ The mechanism for this is a special format registration call that allows *optional* format information to be specified along with the registration. The optional format information API is generalized for future extensions, though at the time of this writing the only optional information supported is the XML markup information. The API for registering a format with optional information and for retrieving the optional information associated with a format is shown below:

```
typedef struct _FMOptInfo {
    int info_type;
    int info_len;
    char *info_block;
} FMOptInfo;

extern FMFormat register_opt_format(const char *format_name, FMFieldList field_list,
                                   FMOptInfo *optinfo, FMContext fmcontext);
extern void * get_optinfo_IOFormat(FMFormat ioformat, int info_type, int *len_p);
```

The `optinfo` parameter to `register_opt_format()` is a list of optional information values to be associated with the format. The list is terminated by an entry with an `info_type` value of 0. To specify optional XML markup information, the `info_type` value must be `XML_OPT_INFO`, the `info_block` value should point to an XML markup string, and the `info_len` parameter should be set to the length of the markup string. The format of the markup string is described in the next sections.

9.2.1 Basic XML Markup

The XML markup string is essentially a template into which the values from the actual data will be substituted. The substitution points are marked by XML-like `FFS:data` tags. Those tags contain as an attribute the identity of the particular data field that is to be substituted, specified by either `field_name` or `field_id` (a zero-based index into the field list). Except for `FFS:*` tags, *all* elements of the XML markup string are copied directly from the template into the hydrated XML. Thus, any non-variant elements in the XML, including tags, constant attributes, or whitespace, can be supplied with the XML markup. For example, an XML markup string that will produce exactly the default output for the example above is:

⁷Note that this approach means that XML formatting is controlled at the time of encoding of data, not at the point at which it is hydrated into XML.

```
"<dynamic format>
<string field><FFS:data field_name="string field"></string field>
<icount><FFS:data field_name=icount></icount>
<double_array>
<FFS:data field_id=2></double_array>
</dynamic format>"
```

Note the variations in the attribute associated with each `FFS:data` tag. Field names are used for the first two tags, and in the case of `string field` the name must be surrounded by quotes because it contains a space character. The last field is specified by its zero-based index in the field list (“`double_array`” is field 2).

```
"<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE dynamic [
  <!ELEMENT dynamic (strfield, icount, doublearray)>
  <!ELEMENT strfield (#PCDATA)>
  <!ELEMENT icount (#PCDATA)>
  <!ELEMENT doublearray (#PCDATA)>
]>
<dynamic>
  <strfield><FFS:data field_name="string field"></strfield>
  <icount><FFS:data field_name=icount></icount>
  <doublearray><FFS:data field_name=double_array></doublearray>
</dynamic>"
```

A program that demonstrates this facility and its output appear in Figures 6 and 7.

9.2.2 XML Markup for Arrays

The simple template mechanism discussed is perfectly adequate for simple and nested structures, but doesn’t allow complete control of how arrays are hydrated into XML. In particular, the use of a simple `FFS:data` tag doesn’t allow control of the text that is to appear before and after each array element. In order to attain that control, we again follow an XML style in the template and introduce two new tags, `FFS:array` and `FFS:array_data_tag`. Essentially, where the basic use of `FFS:data` produces a template with this conceptual behavior:

```
...<FFS:data field_name=field_a>
  text between field_a and field_b (appears once)
<FFS:data field_name=field_b>
  text between field_b and field_c (appears once)
<FFS:data field_name=field_c> ...
```

The new tags define the text in this manner:

```
...<FFS:data field_name=field_a>
  text between field_a and start of array (appears once)
<FFS:array>
  text that appears before every array element
<FFS:array_data_tag field_name=field_b>
  text that appears after every array element
</FFS:array>
  text after array and before field_c (appears once)
<FFS:data field_name=field_c> ...
```

```

#include "io.h"
typedef struct _dyn_rec {
    char      *string;
    long      icount;
    double    *double_array;
} dyn_rec, *dyn_rec_ptr;

char markup[] = "\
<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>\n\
<!DOCTYPE dynamic [\n\
    <!ELEMENT dynamic (strfield, icount, doublearray)>\n\
    <!ELEMENT strfield (#PCDATA)>\n\
    <!ELEMENT icount (#PCDATA)>\n\
    <!ELEMENT doublearray (#PCDATA)>\n\
]>\n\
<dynamic>\n\
    <strfield><PBIO:data field_name=\"string field\"></strfield>\n\
    <icount><PBIO:data field_name=icount></icount>\n\
    <doublearray><PBIO:data field_name=double_array></doublearray>\n\
</dynamic>";

IOField dyn_field_list[] = {
    {"string field", "string", sizeof(char *), IOOffset(dyn_rec_ptr, string)},
    {"icount", "integer", sizeof(long), IOOffset(dyn_rec_ptr, icount)},
    {"double_array", "float[icount]", sizeof(double), IOOffset(dyn_rec_ptr, double_array)},
    { NULL, NULL, 0, 0}
};

int main()    /* sending program */
{
    IOContext src_context = create_IOcontext();
    IOFormat dyn_rec_ioformat;
    dyn_rec rec;
    int buf_size, fd, i;
    char *encoded_buffer, *xml_hydration;
    IOOptInfo opt_info[2];

    opt_info[0].info_type = XML_OPT_INFO;
    opt_info[0].info_len = strlen(markup);
    opt_info[0].info_block = markup;
    opt_info[1].info_type = 0;
    dyn_rec_ioformat = register_opt_format("dynamic format", dyn_field_list, opt_info, src_context);
    rec.string = "Hi Mom!";
    rec.icount = 5;
    rec.double_array = (double*) malloc(sizeof(double) * 5);
    for (i=0; i<5; i++) rec.double_array[i] = i*2.717;
    encoded_buffer = encode_IOcontext_buffer(src_context,
        dyn_rec_ioformat, &rec, &buf_size);

    xml_hydration = IOencoded_to_XML_string(src_context, encoded_buffer);
    printf("encoded data is %d bytes, XML string is %d bytes\n", buf_size, strlen(xml_hydration));
    printf("XML string is :\n%s", xml_hydration);
}

```

Figure 6: A program using explicit XML markup specified as optional information in format registration.

```

encoded data is 80 bytes, XML string is 341 bytes
XML string is :
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE dynamic [
    <!ELEMENT dynamic (strfield, icount, doublearray)>
    <!ELEMENT strfield (#PCDATA)>
    <!ELEMENT icount (#PCDATA)>
    <!ELEMENT doublearray (#PCDATA)>
]>
<dynamic>
    <strfield>Hi Mom! </strfield>
    <icount>5 </icount>
    <doublearray>0 2.717 5.434 8.151 10.868 </doublearray>
</dynamic>

```

Figure 7: Output for the program in the previous figure.

This is sufficient to allow complete control if the representation of arrays in lists in XML hydration. As an example of using this facility, consider the program of Figure 6 and replace the line “<doublearray><FFS:data field_name=double_array></doublearray>” with:

```

<doublearray><FFS:array>
    <list_element>
        <FFS:array_data_mark field_name=double_array>
    </list_element></FFS:array>
</doublearray>

```

The modified portion of the output of the program is shown in Figure 8.

10 COD

CoD (from C-on-Demand) is a small programming language designed to facilitate dynamic computations, typically performed on data that has been encoded with FFS, or at least described with FFS-based FMfield lists. See [?] for the rationale that prompted the need for dynamic computation. FFS builds upon its use of dynamic code generation for FFS conversion functions, preserving the expressiveness of a general programming language and the efficiency of shared objects while retaining the generality of interpreted languages. Functionality such as *in situ* and in-transit processing are expressed in CoD (C On Demand), a subset of a general procedural language, and dynamic code generation is used to create a native version of these functions on the host where the function must be executed. CoD is currently a subset of C, supporting the standard C operators and control flow statements.

Like the DCG used for FFS format conversions, CoD’s dynamic code generation capabilities are based on the Georgia Tech DILL package, to which are added a lexer, parser, semanticizer, and code generator. The CoD/Dill system is a library-based compilation system that generates native machine code directly into the application’s memory without reference to an external compiler, assembler or linker. Only minimal optimizations and basic register allocation are performed, but the resulting native code still significantly outperforms interpreted approaches.

10.1 The CoD Language

CoD may be extended as future needs warrant, but currently it is a subset of C. Currently it supports the C operators, function calls, `for` loops, `if` statements and `return` statements.

```

<doublearray>
  <list_element>
    0
  </list_element>
  <list_element>
    2.717
  </list_element>
  <list_element>
    5.434
  </list_element>
  <list_element>
    8.151
  </list_element>
  <list_element>
    10.868
  </list_element>
</doublearray>

```

Figure 8: New output with array expansion explicitly controlled.

In terms of basic types, it supports C-style declarations of integer and floating point types. That is, the type specifiers `char`, `short`, `int`, `long`, `signed`, `unsigned`, `float` and `double` may be mixed as in C. `void` is also supported as a return type. CoD does not currently support pointers, though the type `string` is introduced as a special case with limited support.

CoD supports the C operators `!,+,-,*,/,%,>,(<,<=,>=,==,!=,&&,&&||,=` with the same precedence rules as in C. Parenthesis can be used for grouping in expressions. As in C, assignment yields a value which can be used in an expression. C's pre/post increment/decrement operators are not included in CoD. Type promotions and conversions also work as in C.

As in C, semicolons are used to end statements and brackets are used to group them. Variable declarations must precede executable statements. So, a simple CoD segment would be:

```

{
  int j = 0;
  long k = 10;
  double d = 3.14;

  return d * (j + k);
}

```

10.2 Generating Code

The subroutine which translates CoD to native code is `cod_code_gen()`. The sample program below illustrates a very simple use of `cod_code_gen()` using the previous CoD segment.

```

#include "cod.h"

char code_string[] = "\
{\n\
    int j = 4;\n\
    long k = 10;\n\
    short l = 3;\n\
\n\
    return l * (j + k);\n\
}";

main()
{
    cod_parse_context context = new_cod_parse_context();
    cod_code gen_code;
    long (*func)();

    gen_code = cod_code_gen(code_string, context);
    func = (long(*)()) gen_code->func;

    printf("generated code returns %ld\n", func());
    cod_free_parse_context(context);
    cod_code_free(gen_code);
}

```

When executed, this program should print “generated code returns 42.” Note that code generation creates a function in malloc’d memory and returns a pointer to that function. That pointer is cast into the appropriate function pointer type and stored in the `gen_code` variable. It is the programs responsibility to free the function memory when it is no longer needed. This demonstrates basic code generation capability, but the functions generated are not particularly useful. The next sections will extend these basic capabilities.

10.3 Parameters and Structured Types

10.3.1 Simple Parameters

The simplest extensions to the subroutine generated above involve adding parameters of atomic data types. As an example, we’ll add an integer parameter “i”. To make use of this parameter we’ll modify the return expression in the `code_string` to `l * (j + k + i)`. The subroutine `cod_add_param()` is used to add the parameter to the generation context. It’s parameters are the name of the parameter, it’s data type (as a string), the parameter number (starting at zero) and the `cod_parse_context` variable. To create a function with a prototype like `long f(int i)` the code becomes:

```

    cod_parse_context context = new_cod_parse_context();
    cod_code gen_code;
    long (*func)(int i);

    cod_add_param("i", "int", 0, context);
    gen_code = cod_code_gen(code_string, context);
    func = (long(*)()) gen_code->func;

    printf("generated code returns %ld\n", func(15));
    cod_free_parse_context(context);
    cod_code_free(gen_code);
}

```

When executed, this program prints ‘‘generated code returns 87.’’ Additional parameters of atomic data types can be added in a similar manner.

As a simpler alternative to multiple calls to `cod_add_param()`, and to provide a mechanism for specifying the return type of the generated routine, CoD also provides :

```
extern void
cod_subroutine_declaration(const char *decl, cod_parse_context context);
```

`cod_subroutine_declaration()` allows the full profile of the generated routine to be specified at once. So, the simple example code above can be modified as below to use this call:

```
cod_subroutine_declaration("int proc(int i)", context);
gen_code = cod_code_gen(code_string, context);
func = (long(*)()) gen_code->func;
```

10.3.2 Structured Types

Adding structured and array parameters to the subroutine is slightly more complex, but builds upon the structure declarations used in FFS. For example consider the structure `test_struct` and its FMField list as below:

```
typedef struct test {
    int i;
    int j;
    long k;
    short l;
} test_struct, *test_struct_p;

ecl_field_elem struct_fields[] = {
    {"i", "integer", sizeof(int), I00ffset(test_struct_p, i)},
    {"j", "integer", sizeof(int), I00ffset(test_struct_p, j)},
    {"k", "integer", sizeof(long), I00ffset(test_struct_p, k)},
    {"l", "integer", sizeof(short), I00ffset(test_struct_p, l)},
    {(void*)0, (void*)0, 0, 0}};
```

The `cod_add_struct_type()` routine is used to add this structured type definition to the parse context. If we define a single parameter ‘‘input’’ as this structured type, the new return value in `code_string` is `input.l * (input.j + input.k + input.i)` and the program body is:

```

main() {
    cod_parse_context context = new_cod_parse_context();
    test_struct str;
    cod_code gen_code;
    long (*func)(test_struct *s);

    cod_add_struct_type("struct_type", struct_fields, context);
    cod_add_param("input", "struct_type", 0, context);

    gen_code = cod_code_gen(code_string, context);
    func = (long(*)()) gen_code->func;

    str.i = 15;
    str.j = 4;
    str.k = 10;
    str.l = 3;
    printf("generated code returns %ld\n", func(&str));
    cod_code_free(gen_code);
    cod_free_parse_context(context);
}

```

Note that the structured type parameter is passed by reference to the generated routine. However in `code_string` fields are still referenced with the `'.'` operator instead of `'-'` (which is not present in CoD).

10.4 External Context and Function Calls

With the exception of a few built-in subroutine calls, CoD routines have no ability to reference external routines or data. That is, the only items visible to them are their own parameters. However, applications can add to the set of visible extern entities by declaring those items to CoD and providing their addresses. For example, to make a subroutine accessible in CoD requires defining both the subroutine's profile (return and parameter types) and its address. To specify the subroutine profile, CoD parses C-style subroutine and function declarations, such as:

```

int printf(string format, ...);
void external_proc(double value);

```

This is done through the subroutine `cod_parse_for_context()`.

However, to associate addresses with symbols requires a somewhat different mechanism. To accomplish this, CoD allows a list of external symbols to be associated with the `codl_parse_context()` value. This list is simply a null-terminated sequence of $\langle symbolname, symbolvalue \rangle$ pairs of type `codl_extern_entry`. The symbol name should match the name used for the subroutine declaration. For example, the following code sequence makes the subroutine "printf" accessible in CoD:

```

extern int printf();
static ecl_extern_entry externs[] =
{
    {"printf", (void*)printf},
    {NULL, NULL}
};

static char extern_string[] = "int printf(string format, ...);";

{
    ecl_parse_context context = new_ecl_parse_context();
    ecl_assoc_externs(context, externs);
    ecl_parse_for_context(extern_string, context);
    ...
}

```

Note that some compiler/OS combinations might disallow initializing a data item with the address of a subroutine (as the second item in the `externs[0]` initialization) because some linkers won't resolve symbolic references in data segments. In this case, the `externs[0].extern_value` must be initialized with an assignment statement.

10.5 'Magic' Assignments and Memory Allocation

Section 3.3 introduced FFS' dynamic array capability, in which an integer-typed field in a structure is identified as the size of a dynamic array in that structure. CoD directly supports this capability in that if field A is the size value of some dynamic array, and A appears on the left hand side of an assignment statement, CoD will automatically `realloc()` the dynamic array to correspond to the new size value. Generally, this is the preferred memory allocation mechanism in CoD because it's nicely type-safe. However, one has to be a bit careful with it. In this circumstance in particular, it is important to make sure that input data to CoD has been reasonably initialized (for example using `memset()` to zeros). Because if that field A and the pointer to the dynamic array have some random garbage value, then an assignment to field A will cause the random garbage pointer to be submitted to `realloc()`, likely causing a segmentation fault.

`malloc()` is not generally a visible operation in CoD because whether or not it is appropriate or safe depends upon the environment in which CoD is used. If CoD is deployed as a processing infrastructure inside a middleware, using `malloc()` may inevitably leak memory because the middleware has no mechanism for tracking and deallocating the created blocks. The details of memory tracking and allowable memory allocation mechanisms will be left to the manuals of that middleware.

10.6 Operating on Unknown Data

All of the examples presented so far in this manual involve data structures that are statically known to the application. However, this has been done for simplicity in the programs and in the exposition, not because FFS/CoD has any restriction or preference for operating only with structures that are known at compile-time. In fact, FFS and CoD do not rely upon any compile-time knowledge of the data structures/messages that they operate upon. The static FMField lists in the examples can be replaced with dynamically generated field lists without invalidating any operations.

10.6.1 FFS-encoded Parameters

There are a several approaches to operating on data whose exact nature is not known to the application at compile-time. One of the simplest is to use CoD's ability to operate directly upon FFS-encoded data. For example, consider this modified program, modified from Section 4.2:

```

int main()      /* receiving program */
{
    FFSTypeHandle second_rec_handle;
    FMContext fmc = create_FMcontext();
    int fd, encode_size;
    char encoded_buffer[2048]; /* hopefully big enough */

    /* "receive" encoded record over a file */
    fd = open("enc_file", O_RDONLY, 0777);
    encode_size = read(fd, encoded_buffer, sizeof(encoded_buffer));

    cod_add_encoded_param("input", encoded_buffer, fmc, context);

    gen_code = cod_code_gen("{return input.j", context);
    if (!gen_code) {
        printf("The input did not have an acceptable field 'j'\n");
    } else {
        func = (long(*)()) gen_code->func;
        printf("Field 'j' in the input has value %d\n", func(encoded_buffer));
    }
}

```

Here, we use the FFS routine `cod_add_encoded_param()`, both adds a structured type (the type of the actual encoded data) and a parameter of that type. In this case we give the parameter the name 'input'. The call to `cod_code_gen()` will succeed if the incoming data has a field named 'j' of an acceptable type (I.E. not an array, a pointer, etc.). If the `cod_code_gen()` call succeeds, then the generated function (here, `func` expects a pointer to the encoded buffer as its first parameter. Note that while we do this code generation using an exemplar data block, once generated the subroutine `func` can be used with any data block with the same FFS format ID.⁸

The approach of operating on FFS-encoded values has the advantage of simplicity and efficiency. In particular, the incoming data is not transformed into a local native format, but instead values are converted upon demand. This can be a considerable efficiency if only a fraction of the incoming data is referenced. However, it also has significant limitations. In particular, the CoD code must not make changes in the parameter. I.E. it shouldn't be on the left hand side of an assignment, be indirectly operated on via pointers, etc. (Strictly speaking, CoD should declare it `const`, but CoD does not currently have `const` support.)

10.6.2 Operating on data after decoding to 'local' formats

There is a somewhat more complex alternative approach that allows generated CoD routines to make changes in the incoming data. Structurally, the approach is quite similar to the decoding program of Section 4.2, but if we're dealing with data about which we have no *a priori* knowledge, we don't have the information necessary to perform the call to `FFSset_simple_target()`. So, the first step is to acquire this information. Several routines will be useful here:

```

extern FFSTypeHandle
FFSTypeHandle_from_encode(FFSContext c, char *b);

extern FMFormat
FMFormat_of_original(FFSTypeHandle h);

```

⁸A more complete program would keep track of format IDs for which it has generated a routine, storing them in a data structure for later re-use. Upon encountering new data, it would extract the FFS format ID from the data, compare it to those that were stored and attempt new code generation if not found. This program is left as an exercise for the reader.

```
extern FMStructDescList
get_localized_formats(FMFormat f);
```

Of these, `FFSTypeHandle_from_encode()` causes FFS to extract the format ID from the encoded data, query the format server to acquire full meta-information about the data and return a handle to the information. `FMFormat_of_original()` returns the format information as it was provided by the *original encoder* of the data. This encoder-side format information may be incompatible with the data representation in use by the receiver. For example, pointer sizes, data alignment requirements, and other important aspects of data representation may differ. The FFS routine `get_localized_formats()` will extract the `FMStructDescList` of the encoder and massage it so that the structure descriptions are acceptable to the receiver. At this point, these modified format lists can be used in a call to `FFSset_simple_target()` as below;

```
int main()      /* receiving program */
{
    FFSTypeHandle handle;
    FMFormat fmf;
    FMStructDescList local_struct;
    FFSContext ffsc = create_FFSContext();
    int fd, encode_size, decode_size;
    char encoded_buffer[2048]; /* hopefully big enough */
    void *unknown_record;
    cod_parse_context context = new_cod_parse_context();
    cod_code gen_code;
    int (*func)(void *);

    /* "receive" encoded record over a file */
    fd = open("enc_file", O_RDONLY, 0777);
    encode_size = read(fd, encoded_buffer, sizeof(encoded_buffer));

    handle = FFSTypeHandle_from_encode(ffsc, encoded_buffer);
    fmf = FMFormat_of_original(handle);
    local_struct = get_localized_formats(fmf);

    /* decode the data into an acceptable local format */
    FFSset_fixed_target(ffsc, local_struct);
    FFS_target_from_encode(ffsc, encoded_buffer);
    decode_size = FFS_est_decode_length(ffsc, encoded_buffer, encode_size);
    unknown_record = malloc(decode_size);
    FFSdecode(ffsc, encoded_buffer, unknown_record);

    /* generate code to operate on it */
    cod_add_struct_type(local_struct, context);
    cod_add_param("input", FFSTypeHandle_name(handle), 0, context);
    gen_code = cod_code_gen("{return input.j", context);
    if (!gen_code) {
        printf("The input did not have an acceptable field 'j'\n");
    } else {
        func = (long(*)()) gen_code->func;
        printf("Field 'j' in the input has value %d\n", func(unknown_record));
    }
}
```

The two just preceding code segments have demonstrated some of the abilities that FFS and CoD have in operating on unknown data. However, there are still dangers in combining some types of operations.

Our experience with FFS has shown that most programs do not operate upon data that they have no prior knowledge about. While FFS `IOFieldLists` can be created and used dynamically, it is a rare practice

because it is only necessary if the structures to be manipulated are dynamic and that is not a semantic that is directly supported by common programming environments. Therefore, FFSs routines that query and manipulate field lists will not be described in detail here. Instead we will just enumerate them below:

`extern IOFieldList field_list_of_IOformat(IOFormat format)` returns the NULL-terminated `IOFieldList` associated with a given

`extern int compare_field_lists(IOFieldList list1, IOFieldList list2)` compares two field lists for strict equality.

`extern IOFieldList copy_field_list(IOFieldList list)` returns a copy of an `IOFieldList`

`extern IOFieldList localize_field_list(IOFieldList list)` this routine assigns “reasonable” values to the size and offset values for the given field list. Here “reasonable” means acceptable to the current underlying machine architecture. This routine is used to support higher-level dynamic code generation routines in doing third-party data processing and filtering.

`extern int struct_size_field_list(IOFieldList list, int pointer_size)` returns the size of the “base” structure described by a field list. The `pointer_size` parameter is required because that information is implicit in format registration and not carried with the field list.

11 Standard Tools for FFS Files

The meta-information contained in a FFS data stream allows the construction of general tools to operate on FFS files. Two such tools are provided with the FFS library, `FFSdump` and `FFSsort`.

`FFSdump` is a “cat” program for FFS files. `FFSdump` takes as arguments a set of options and a filename. By default `FFSdump` prints an ascii representation of all data records and comments in the FFS file. Dumping of record format information as well as header information in the file is enabled by specifying options of `+formats` and `+header` respectively. In general, printing for any record type can be turned on or off by specifying options of the form `{+,-}{header, comments, formats, data}`.

`FFSsort` is a generalized sorting program for FFS files. It takes three parameters, the name of the field to sort upon, the name of the input file and the name of the output file. The sort field can be of any FFS atomic data type, but it must be the same basic type in all records. Any records in which the sort field does not appear will not appear in the output file.

References

- [1] Hasan Abbasi, Matthew Wolf, Fang Zheng, Greg Eisenhauer, Scott Klasky, and Karsten Schwan. Scalable data staging services for petascale applications. In *Proceedings of the International Symposium on High Performance Distributed Computing 2009 (HPDC'09)*, June 2009.
- [2] Sandip Agarwala, Greg Eisenhauer, and Karsten Schwan. Lightweight morphing support for evolving data exchanges in distributed applications. In *Proc. of the 25th International Conference on Distributed Computer Systems (ICDCS-25)*, June 2005.