1. License.

2. Introduction.

3. About this library. This is Marpa's "ami" or "friend" library, for macros and functions which are useful for Libmarpa and its "close friends". The contents of this library are considered "undocumented", in the sense that they are not documented for general use. Specifically, the interfaces of these functions is subject to radical change without notice, and it is assumed that the safety of such changes can be ensured by checking only Marpa itself and its "close friend" libraries.

A "close friend" library is one which is allowed to rely on undocumented Libmarpa interfaces. At this writing, the only example of a "close friend" library is the Perl XS code which interfaces libmarpa to Perl.

The ami interface and an internal interface differ in that

- The ami interface must be useable in a situation where the Libmarpa implementor does not have complete control over the namespace. It can only create names which begin in `marpa_`, `_marpa_` or one of its capitalization variants. The internal interface can assume that no library will be included unless the Libmarpa implementor decided it should be, so that most names are available for his use.
- The ami interface cannot use Libmarpa's error handling – although it can be part of the implementation of that error handling. The ami interface must be useable in a situation where another error handling regime is in effect.

4. About this document. This document is very much under construction, enough so that readers may question why I make it available at all. Two reasons:

- Despite its problems, it is the best way to read the source code at this point.
- Since it is essential to changing the code, not making it available could be seen to violate the spirit of the open source.

5. Inlining. Most of this code in `libmarpa` will be frequently executed. Inlining is used a lot. Enough so that it is useful to define a macro to let me know when inlining is not used in a private function.

⟨ Private macros 5 ⟩ ≡
```
#define PRIVATE_NOT_INLINE static
#define PRIVATE static inline
```
This code is used in section 32.

6. Memory allocation. libmarpa wrappers the standard memory functions to provide more convenient behaviors.

- The allocators do not return on failed memory allocations.
- `my_realloc` is equivalent to `my_malloc` if called with a Λ pointer. (This is the GNU C library behavior.)

7. To Do: For the moment, the memory allocators are hard-wired to the C89 default `malloc` and `free`. At some point I may allow the user to override these choices.

⟨ Friend static inline functions 7 ⟩ ≡

```
static inline void my_free(void *p)
{
  free(p);
}
```

See also sections 8, 16, and 18.

This code is used in section 28.

8. ⟨ Friend static inline functions 7 ⟩ +≡

```
static inline void *my_malloc(size_t size)
{
  void *newmem ⟸ malloc(size);
  if (_MARPA_UNLIKELY(¬newmem)) {
    (*marpa__out_of_memory)( );
  }
  return newmem;
}
static inline void *my_malloc0(size_t size)
{
  void *newmem ⟸ my_malloc(size);
  memset(newmem, 0, size);
  return newmem;
}
static inline void *my_realloc(void *p, size_t size)
{
  if (_MARPA_LIKELY(p ≠ Λ)) {
    void *newmem ⟸ realloc(p, size);
    if (_MARPA_UNLIKELY(¬newmem)) (*marpa__out_of_memory)( );
    return newmem;
  }
  return my_malloc(size);
}
```

9.

```
#define marpa_new(type, count)
        ( ( type * ) my_malloc((sizeof (type) * ((size_t)(count)))) )
#define marpa_renew(type, p, count)
        ( ( type * ) my_realloc((p), (sizeof (type) * ((size_t)(count)))) )
```

10. Dynamic stacks. `libmarpa` uses stacks and worklists extensively. This stack interface resizes itself dynamically. There are two disadvantages.

- There is more overhead — overflow must be checked for with each push, and the resizings, while fast, do take time.
- The stack may be moved after any `MARPA_DSTACK_PUSH` operation, making all pointers into it invalid. Data must be retrieved from the stack before the next `MARPA_DSTACK_PUSH`. In the special 2-argument form, `MARPA_DSTACK_INIT2`, the stack is initialized to a size convenient for the memory allocator. **To Do**: Right now this is hard-wired to 1024, but I should use the better calculation made by the obstack code.

```
#define MARPA_DSTACK_DECLARE(this) struct marpa_dstack_s this
#define MARPA_DSTACK_INIT(this, type, initial_size)
    (((this).t_count ⟸ 0), ((this).t_base ⟸ marpa_new(type,
        ((this).t_capacity ⟸ (initial_size)))))
#define MARPA_DSTACK_INIT2(this, type)
    MARPA_DSTACK_INIT((this), type, MAX(4, 1024/sizeof (this)))
```

11. `MARPA_DSTACK_SAFE` is for cases where the dstack is not immediately initialized to a useful value, and might never be. All fields are zeroed so that when the containing object is destroyed, the deallocation logic knows that no memory has been allocated and therefore no attempt to free memory should be made.

```
#define MARPA_DSTACK_IS_INITIALIZED(this) ((this).t_base)
#define MARPA_DSTACK_SAFE(this)
    (((this).t_count ⟸ (this).t_capacity ⟸ 0), ((this).t_base ⟸ Λ))
```

12. It is up to the caller to ensure that there is sufficient capacity for the new count. Usually this call will be used to shorten the stack, in which case capacity is not an issue.

```
#define MARPA_DSTACK_COUNT_SET(this, n) ((this).t_count ⟸ (n))
```

13. A stack reinitialized by `MARPA_DSTACK_CLEAR` contains 0 elements, but has the same capacity as it had before the reinitialization. This saves the cost of reallocating the dstack's buffer, and leaves its capacity at what is hopefully a stable, high-water mark, which will make future resizings unnecessary.

```
#define MARPA_DSTACK_CLEAR(this) MARPA_DSTACK_COUNT_SET((this), 0)
#define MARPA_DSTACK_PUSH(this, type) ((_MARPA_UNLIKELY((this).t_count ≥
        (this).t_capacity) ? marpa_dstack_resize2(&(this), sizeof (type)) : 0),
        ( ( type * ) (this).t_base + (this).t_count++ ) )
#define MARPA_DSTACK_POP(this, type)
    ( (this).t_count ≤ 0 ? Λ : ( ( type * ) (this).t_base + (--(this).t_count) ) )
#define MARPA_DSTACK_INDEX(this, type, ix)
    (MARPA_DSTACK_BASE((this), type) + (ix))
#define MARPA_DSTACK_TOP(this, type)
    (MARPA_DSTACK_LENGTH(this) ≤ 0 ? Λ : MARPA_DSTACK_INDEX((this), type,
        MARPA_DSTACK_LENGTH(this) − 1))
```

```
#define MARPA_DSTACK_BASE(this,type) ( ( type * ) (this).t_base )
#define MARPA_DSTACK_LENGTH(this) ((this).t_count)
#define MARPA_DSTACK_CAPACITY(this) ((this).t_capacity)
```

14. *DSTACK*'s can have their data "stolen", by other containers. The `MARPA_STOLEN_DSTACK_DATA_FREE` macro is intended to help the "thief" container deallocate the data it now has "stolen".

```
#define MARPA_STOLEN_DSTACK_DATA_FREE(data) (my_free(data))
#define MARPA_DSTACK_DESTROY(this)
    MARPA_STOLEN_DSTACK_DATA_FREE(this.t_base)
```

⟨ Friend incomplete structures 14 ⟩ ≡

```
struct marpa_dstack_s;
typedef struct marpa_dstack_s *MARPA_DSTACK;
```

This code is used in section 27.

15. ⟨ Friend structures 15 ⟩ ≡

```
struct marpa_dstack_s {
  int t_count;
  int t_capacity;
  void *t_base;
};
```

This code is used in section 28.

16. ⟨ Friend static inline functions 7 ⟩ +≡

```
static inline void *marpa_dstack_resize2(struct marpa_dstack_s *this, int
    type_bytes)
{
  return marpa_dstack_resize(this, type_bytes, this→t_capacity * 2);
}
```

17.

```
#define MARPA_DSTACK_RESIZE(this,type,new_size)
    (marpa_dstack_resize((this), sizeof (type), (new_size)))
```

18. ⟨ Friend static inline functions 7 ⟩ +≡

```
static inline void *marpa_dstack_resize(struct marpa_dstack_s *this, int
    type_bytes, int new_size)
{
  if (new_size > this→t_capacity) {
      /* We do not shrink the stack in this method */
    this→t_capacity ⟸ new_size;
    this→t_base ⟸ my_realloc(this→t_base, (size_t) new_size * (size_t)
        type_bytes);
  }
  return this→t_base;
}
```

19. Debugging. The MARPA_DEBUG flag enables intrusive debugging logic. "Intrusive" debugging includes things which would be annoying in production, such as detailed messages about internal matters on STDERR. MARPA_DEBUG is expected to be defined in the CFLAGS. MARPA_DEBUG implies MARPA_ENABLE_ASSERT, but not vice versa.

⟨ Debug macros 19 ⟩ ≡
```
#define MARPA_OFF_DEBUG1  (a)
#define MARPA_OFF_DEBUG2  (a, b)
#define MARPA_OFF_DEBUG3  (a, b, c)
#define MARPA_OFF_DEBUG4  (a, b, c, d)
#define MARPA_OFF_DEBUG5  (a, b, c, d, e)
#define MARPA_OFF_ASSERT  (expr)
```
See also section 21.

This code is used in section 27.

20. Returns int so that it can be portably used in a logically-anded expression.

⟨ Function definitions 20 ⟩ ≡
```
int marpa__default_debug_handler(const char *format, ...)
{
  va_list args;
  va_start(args, format);
  vfprintf(stderr, format, args);
  va_end(args);
  putc('\n', stderr);
  return 1;
}
```
This code is used in section 33.

21. ⟨ Debug macros 19 ⟩ +≡
```
#ifndef MARPA_DEBUG
#define MARPA_DEBUG  0
#endif
#if MARPA_DEBUG
#define MARPA_DEBUG1(a)  (void)(marpa__debug_level ∧ (*marpa__debug_handler)(a))
#define MARPA_DEBUG2(a, b)
  (void)(marpa__debug_level ∧ (*marpa__debug_handler)((a), (b)))
#define MARPA_DEBUG3(a, b, c)
  (void)(marpa__debug_level ∧ (*marpa__debug_handler)((a), (b), (c)))
#define MARPA_DEBUG4(a, b, c, d)
  (void)(marpa__debug_level ∧ (*marpa__debug_handler)((a), (b), (c), (d)))
#define MARPA_DEBUG5(a, b, c, d, e)
  (void)(marpa__debug_level ∧ (*marpa__debug_handler)((a), (b), (c), (d), (e)))
#else
#define MARPA_DEBUG1  (a)
#define MARPA_DEBUG2  (a, b)
#define MARPA_DEBUG3  (a, b, c)
```

```
#define MARPA_DEBUG4 (a,b,c,d)
#define MARPA_DEBUG5 (a,b,c,d,e)
#endif
#if MARPA_DEBUG
#undef MARPA_ENABLE_ASSERT
#define MARPA_ENABLE_ASSERT 1
#endif
#ifndef MARPA_ENABLE_ASSERT
#define MARPA_ENABLE_ASSERT 0
#endif
#if MARPA_ENABLE_ASSERT
#undef MARPA_ASSERT
#define MARPA_ASSERT(expr) do {
    if (_MARPA_LIKELY (expr));
    else (*marpa__debug_handler)("%s: assertion failed %s", STRLOC, #expr);
  } while (0);
#else /* if not MARPA_DEBUG */
#define MARPA_ASSERT(exp)
#endif
```

22. Internal macros.

⟨ Internal macros 22 ⟩ ≡

```
#if __GNUC__ > 2 ∨ (__GNUC__ ≡ 2 ∧ __GNUC_MINOR__ > 4)
#define UNUSED__attribute__ ((__unused__))
#else
#define UNUSED
#endif
#if defined (__GNUC__) ∧ defined (__STRICT_ANSI__)
#undef inline
#define inline __inline__
#endif
#undef Dim
#define Dim(x) (sizeof (x)/sizeof (*x))
#undef MAX
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
#undef CLAMP
#define CLAMP(x,low,high) (((x) > (high)) ? (high) : (((x) < (low)) ? (low) : (x)))
#undef STRINGIFY_ARG
#define STRINGIFY_ARG(contents)#contents
#undef STRINGIFY
#define STRINGIFY(macro_or_string)STRINGIFY_ARG (macro_or_string)
    /* A string identifying the current code position */
#if defined (__GNUC__) ∧ (__GNUC__ < 3) ∧ ¬defined (__cplusplus)
#define STRLOC __FILE__":"STRINGIFY(__LINE__)":"__PRETTY_FUNCTION__ "()"
#else
```

```
#define STRLOC__FILE__":"STRINGIFY (__LINE__)
#endif /* Provide a string identifying the current function, non-concatenatable */
#if defined (__GNUC__)
#define STRFUNC ((const char *)(__PRETTY_FUNCTION__))
#elif defined (__STDC_VERSION__) ∧ __STDC_VERSION__ ≥ 19901L
#define STRFUNC ((const char *)(__func__))
#else
#define STRFUNC ((const char *)("???"))
#endif
#if defined __GNUC__
#define alignof(type) (__alignof__(type))
#else
#define alignof(type) (offsetof(struct {
    char __slot1;
    type __slot2;
  }, __slot2))
#endif
```

This code is used in sections 23 and 27.

23. Internal typdefs.

⟨ Internal typedefs 23 ⟩ ≡

```
typedef unsigned int BITFIELD; ⟨ Internal macros 22 ⟩
#define Boolean(value) ((value) ? 1 : 0)
```

This code is used in section 27.

24. File layout.

25. The output files are written in pieces, with the license prepended, which allows it to start the file. The output files are **not** source files, but I add the license to them anyway.

26. Also, it is helpful to someone first trying to orient herself, if built source files contain a comment to that effect and a warning not that they are not intended to be edited directly. So I add such a comment.

27. marpa_ami.h layout, first piece.

```
⟨ marpa_ami.h.p10 27 ⟩ ≡
#ifndef _MARPA_AMI_H__
#define _MARPA_AMI_H__ 1
#if defined (__GNUC__) ∧ (__GNUC__ > 2) ∧ defined (__OPTIMIZE__)
#define _MARPA_LIKELY(expr) (__builtin_expect((expr), 1))
#define _MARPA_UNLIKELY(expr) (__builtin_expect((expr), 0))
#else
#define _MARPA_LIKELY(expr) (expr)
#define _MARPA_UNLIKELY(expr) (expr)
#endif
  ⟨ Debug macros 19 ⟩⟨ Internal macros 22 ⟩⟨ Internal typedefs 23 ⟩
  ⟨ Preprocessor definitions ⟩
  ⟨ Friend incomplete structures 14 ⟩
```

28. marpa_ami.h layout, last piece.

```
⟨ marpa_ami.h.p90 28 ⟩ ≡
  ⟨ Friend structures 15 ⟩
  ⟨ Friend static inline functions 7 ⟩
#endif /* _MARPA_AMI_H__ */
```

29. marpa_ami.c layout.

30.
```
⟨ marpa_ami.c.p10 30 ⟩ ≡
#include "config.h"
```
See also sections 31 and 32.

31. These C90 headers are needed for the default debug handler. This is strictly C90 and is always compiled in. We don't want to require applications to obey the MARPA_DEBUG flag and compile conditionally. This means that applications must be allowed to set the debug level and handler, even when debugging is not compiled in, and they will be meaningless.

```
⟨ marpa_ami.c.p10 30 ⟩ +≡
#include <stdarg.h>
#include <stdio.h>
```

32. ⟨ marpa_ami.c.p10 30 ⟩ +≡
```
#ifndef MARPA_DEBUG
#define MARPA_DEBUG 0
#endif
#include "marpa.h"
#include "marpa_ami.h"
```
⟨ Private macros 5 ⟩

33. The .c file has no contents at the moment, so just in case, I include a dummy function. Once there are other contents, it should be deleted.

⟨ marpa_ami.c.p50 33 ⟩ ≡
⟨ Function definitions 20 ⟩

34. Index.

⟨ Debug macros 19, 21 ⟩ Used in section 27.
⟨ Friend incomplete structures 14 ⟩ Used in section 27.
⟨ Friend static inline functions 7, 8, 16, 18 ⟩ Used in section 28.
⟨ Friend structures 15 ⟩ Used in section 28.
⟨ Function definitions 20 ⟩ Used in section 33.
⟨ Internal macros 22 ⟩ Used in sections 23 and 27.
⟨ Internal typedefs 23 ⟩ Used in section 27.
⟨ Private macros 5 ⟩ Used in section 32.
⟨ `marpa_ami.c.p10` 30, 31, 32 ⟩
⟨ `marpa_ami.c.p50` 33 ⟩
⟨ `marpa_ami.h.p10` 27 ⟩
⟨ `marpa_ami.h.p90` 28 ⟩

Marpa's ami tools

June 13, 2022 at 16:33