Real-world use of X-Macros

Asked 12 years, 4 months ago Modified 9 months ago Viewed 34k times

89

I just learned of X-Macros. What real-world uses of X-Macros have you seen? When are they the right tool for the job?

c macros c-preprocessor x-macros

Share Improve this question Follow

edited Mar 14, 2016 at 23:39
Brian Tompsett - 汤莱恩
5,747 72 58 129

asked Jul 9, 2011 at 15:56
Agnius Vasiliauskas
11k 5 50 70

4 An *Embedded.com* recent series: Part 1, Part 2, Part 3 on this topic. Answers here are as instructive.
– artless noise Mar 30, 2013 at 19:14

For those who are thinking about using X-macros, consider using the following format: `(() () ())` instead of: `( , , )` . This makes them more useful for recursive, variadic macro situations. For more on why/how, see here: stackoverflow.com/a/66130832/1599699 Also, you can avoid using those ugly \'s in your macros by simply putting each entry on its own line and including the file; see here for more: quuxplusone.github.io/blog/2021/02/01/x-macros – Andrew Feb 10, 2021 at 4:25

8 Answers

Sorted by: Highest score (default)

116

I discovered X-macros a couple of years ago when I started making use of function pointers in my code. I am an embedded programmer and I use state machines frequently. Often I would write code like this:

```
/* declare an enumeration of state codes */
enum{ STATE0, STATE1, STATE2, ... , STATEX, NUM_STATES};

/* declare a table of function pointers */
p_func_t jumptable[NUM_STATES] = {func0, func1, func2, ... , funcX};
```

The problem was that I considered it very error prone to have to maintain the ordering of my function pointer table such that it matched the ordering of my enumeration of states.

A friend of mine introduced me to X-macros and it was like a light-bulb went off in my head. Seriously, where have you been all my life x-macros!

So now I define the following table:

```
#define STATE_TABLE \
        ENTRY(STATE0, func0) \
        ENTRY(STATE1, func1) \
        ENTRY(STATE2, func2) \
        ...
        ENTRY(STATEX, funcX) \
```

And I can use it as follows:

```
enum
{
#define ENTRY(a,b) a,
    STATE_TABLE
#undef ENTRY
    NUM_STATES
};
```

and

```
p_func_t jumptable[NUM_STATES] =
{
#define ENTRY(a,b) b,
    STATE_TABLE
#undef ENTRY
};
```

as a bonus, I can also have the pre-processor build my function prototypes as follows:

```
#define ENTRY(a,b) static void b(void);
    STATE_TABLE
#undef ENTRY
```

Another usage is to declare and initialize registers

```
#define IO_ADDRESS_OFFSET (0x8000)
#define REGISTER_TABLE\
    ENTRY(reg0, IO_ADDRESS_OFFSET + 0, 0x11)\
    ENTRY(reg1, IO_ADDRESS_OFFSET + 1, 0x55)\
    ENTRY(reg2, IO_ADDRESS_OFFSET + 2, 0x1b)\
    ...
    ENTRY(regX, IO_ADDRESS_OFFSET + X, 0x33)\

/* declare the registers (where _at_ is a compiler specific directive) */
#define ENTRY(a, b, c) volatile uint8_t a _at_ b:
    REGISTER_TABLE
#undef ENTRY

/* initialize registers */
#define ENTRY(a, b, c) a = c;
    REGISTER_TABLE
#undef ENTRY
```

My favourite usage however is when it comes to communication handlers

First I create a comms table, containing each command name and code:

```
#define COMMAND_TABLE \
    ENTRY(RESERVED,    reserved,    0x00) \
    ENTRY(COMMAND1,    command1,    0x01) \
    ENTRY(COMMAND2,    command2,    0x02) \
    ...
    ENTRY(COMMANDX,    commandX,    0x0X) \
```

I have both the uppercase and lowercase names in the table, because the upper case will be used for enums and the lowercase for function names.

Then I also define structs for each command to define what each command looks like:

```
typedef struct {...}command1_cmd_t;
typedef struct {...}command2_cmd_t;

etc.
```

Likewise I define structs for each command response:

```
typedef struct {...}command1_resp_t;
typedef struct {...}command2_resp_t;

etc.
```

Then I can define my command code enumeration:

```
enum
{
#define ENTRY(a,b,c) a##_CMD = c,
    COMMAND_TABLE
#undef ENTRY
};
```

I can define my command length enumeration:

```
enum
{
#define ENTRY(a,b,c) a##_CMD_LENGTH = sizeof(b##_cmd_t);
    COMMAND_TABLE
#undef ENTRY
};
```

I can define my response length enumeration:

```
enum
{
#define ENTRY(a,b,c) a##_RESP_LENGTH = sizeof(b##_resp_t);
    COMMAND_TABLE
#undef ENTRY
};
```

I can determine how many commands there are as follows:

```
typedef struct
{
#define ENTRY(a,b,c) uint8_t b;
    COMMAND_TABLE
#undef ENTRY
} offset_struct_t;

#define NUMBER_OF_COMMANDS sizeof(offset_struct_t)
```

NOTE: I never actually instantiate the offset_struct_t, I just use it as a way for the compiler to generate for me my number of commands definition.

Note then I can generate my table of function pointers as follows:

```
p_func_t jump_table[NUMBER_OF_COMMANDS] =
{
#define ENTRY(a,b,c) process_##b,
    COMMAND_TABLE
#undef ENTRY
}
```

And my function prototypes:

```
#define ENTRY(a,b,c) void process_##b(void);
    COMMAND_TABLE
#undef ENTRY
```

Now lastly for the coolest use ever, I can have the compiler calculate how big my transmit buffer should be.

```
/* reminder the sizeof a union is the size of its largest member */
typedef union
{
#define ENTRY(a,b,c) uint8_t b##_buf[sizeof(b##_cmd_t)];
    COMMAND_TABLE
#undef ENTRY
}tx_buf_t
```

Again this union is like my offset struct, it is not instantiated, instead I can use the sizeof operator to declare my transmit buffer size.

```
uint8_t tx_buf[sizeof(tx_buf_t)];
```

Now my transmit buffer tx_buf is the optimal size and as I add commands to this comms handler, my buffer will always be the optimal size. Cool!

One other use is to create offset tables: Since memory is often a constraint on embedded systems, I don't want to use 512 bytes for my jump table (2 bytes per pointer X 256 possible commands) when it is a sparse array. Instead I will have a table of 8bit offsets for each possible command. This offset is then used to index into my actual jump table which now only needs to be NUM_COMMANDS * sizeof(pointer). In my case with 10 commands defined. My jump table is 20bytes long and I have an offset table that is 256 bytes long, which is a total of 276bytes instead of 512bytes. I then call my functions like so:

```
jump_table[offset_table[command]]();
```

instead of

```
jump_table[command]();
```

I can create an offset table like so:

```
/* initialize every offset to 0 */
static uint8_t offset_table[256] = {0};

/* for each valid command, initialize the corresponding offset */
#define ENTRY(a,b,c) offset_table[c] = offsetof(offset_struct_t, b);
    COMMAND_TABLE
#undef ENTRY
```

where offsetof is a standard library macro defined in "stddef.h"

As a side benefit, there is a very easy way to determine if a command code is supported or not:

```
bool command_is_valid(uint8_t command)
{
    /* return false if not valid, or true (non 0) if valid */
    return offset_table[command];
}
```

This is also why in my COMMAND_TABLE I reserved command byte 0. I can create one function called "process_reserved()" which will be called if any invalid command byte is used to index into my offset table.

Share Improve this answer Follow   edited Feb 26, 2012 at 18:35   answered Feb 21, 2012 at 20:13

ACRL
1,840 1 13 17

2 Wow! I humbly yield acceptance to this superior answer. (But you should consider the "user-macro" style: no need to undef anything, no need to remember the inner "variable" name.) – luser droog Feb 23, 2012 at 6:07

1 Thanks so much, learned something new today. Now instead of all my #define and #undef I can do the following: REGISTERTABLE(AS_DECLARATION) REGISTERTABLE(AS_INITIALIZER) Very Cool! – ACRL Feb 24, 2012 at 13:47

12 "Seriously, where have you been all my life x-macros!" Lurking in hell, waiting for some unsuspecting programmer to summon them, most likely. In modern C, you can create a direct, tight coupling between the jump table and the enums like this: `p_func_t jumptable[] = { [STATE0] = func0, [STATE1] = func1 };`. Note the `[]` for the array size. Now to ensure no item is missing, add a compile-time check: `_Static_assert(NUM_STATES == sizeof jumptable/sizeof *jumptable, "error");`. Type safe, readable, not a single macro in sight. – Lundin Oct 6, 2016 at 7:50

5 My point here is that x macros should be *the very last resort*, rather than the first thing that springs to mind when you are facing some program design problem. – Lundin Oct 6, 2016 at 7:57

1 embedded.com/design/programming-languages-and-tools/4403953/... – AlphaGoku Mar 23, 2018 at 2:36

46

X-Macros are essentially parameterized templates. So they are the right tool for the job if you need several similar things in several guises. They allow you to create an abstract form and instantiate it according to different rules.

I use X-macros to output enum values as strings. And since encountering it, I strongly prefer this form which takes a "user" macro to apply to each element. Multiple file inclusion is just far more painful to work with.

```
/* x-macro constructors for error and type
   enums and string tables */
#define AS_BARE(a) a ,
#define AS_STR(a) #a ,

#define ERRORS(_) \
    _(noerror) \
    _(dictfull) _(dictstackoverflow) _(dictstackunderflow) \
    _(execstackoverflow) _(execstackunderflow) _(limitcheck) \
    _(VMerror)
enum err { ERRORS(AS_BARE) };
char *errorname[] = { ERRORS(AS_STR) };
/* puts(errorname[(enum err)limitcheck]); */
```

I'm also using them for function dispatch based on object type. Again by hijacking the same macro I used to create the enum values.

```
#define TYPES(_) \
    _(invalid) \
    _(null) \
    _(mark) \
    _(integer) \
    _(real) \
    _(array) \
    _(dict) \
    _(save) \
    _(name) \
    _(string) \
/*enddef TYPES */

#define AS_TYPE(_) _ ## type ,
enum { TYPES(AS_TYPE) };
```

Using the macro guarantees that all my array indices will match the associated enum values, because they construct their various forms using the bare tokens from the macro definition (the TYPES macro).

```
typedef void evalfunc(context *ctx);

void evalquit(context *ctx) { ++ctx->quit; }

void evalpop(context *ctx) { (void)pop(ctx->lo, adrent(ctx->lo, OS)); }

void evalpush(context *ctx) {
    push(ctx->lo, adrent(ctx->lo, OS),
            pop(ctx->lo, adrent(ctx->lo, ES)));
}

evalfunc *evalinvalid = evalquit;
evalfunc *evalmark = evalpop;
evalfunc *evalnull = evalpop;
evalfunc *evalinteger = evalpush;
evalfunc *evalreal = evalpush;
evalfunc *evalsave = evalpush;
evalfunc *evaldict = evalpush;
evalfunc *evalstring = evalpush;
evalfunc *evalname = evalpush;

evalfunc *evaltype[stringtype/*last type in enum*/+1];
#define AS_EVALINIT(_) evaltype[_ ## type] = eval ## _ ;
void initevaltype(void) {
    TYPES(AS_EVALINIT)
}

void eval(context *ctx) {
    unsigned ades = adrent(ctx->lo, ES);
    object t = top(ctx->lo, ades, 0);
    if ( isx(t) ) /* if executable */
        evaltype[type(t)](ctx);  /* <--- the payoff is this line here! */
    else
        evalpush(ctx);
}
```

Using X-macros this way actually helps the compiler to give helpful error messages. I omitted the evalarray function from the above because it would distract from my point. But if you attempt to compile the above code (commenting-out the other function calls, and providing a dummy typedef for context, of course), the compiler would complain about a missing function. For each new type I add, I am reminded to add a handler when I recompile this module. So the X-macro helps to guarantee that parallel structures remain intact even as the project grows.

Edit:

This answer has raised my reputation 50%. So here's a little more. The following is a **negative example**, answering the question: *when **not** to use X-Macros?*

This example shows the packing of arbitrary code fragments into the X-"record". I eventually abandoned this branch of the project and did not use this strategy in later designs (and not for want of trying). It became unweildy, somehow. Indeed the macro is named X6 because at one point there were 6 arguments, but I got tired of changing the macro name.

```
/* Object types */
/* "'X'" macros for Object type definitions, declarations and initializers */
// a                       b            c              d
// enum,                   string,      union member,  printf d
#define OBJECT_TYPES \
X6(    nulltype,        "null",     int dummy      ,            ("<null>")) \
X6(    marktype,        "mark",     int dummy2      ,           ("<mark>")) \
X6( integertype,     "integer",     int  i,     ("%d",o.i)) \
X6( booleantype,     "boolean",     bool b,     (o.b?"true":"false")) \
X6(    realtype,        "real",     float f,        ("%f",o.f)) \
X6(    nametype,        "name",     int  n,     ("%s%s", \
        (o.flags & Fxflag)?"":"/", names[o.n])) \
X6(  stringtype,      "string",     char *s,        ("%s",o.s)) \
X6(    filetype,        "file",     FILE *file,     ("<file %p>",(void *)o.file))
\
X6(   arraytype,       "array",     Object *a,      ("<array %u>",o.length)) \
X6(    dicttype,        "dict",     struct s_pair *d, ("<dict %u>",o.length)) \
X6(operatortype,    "operator",     void (*o)(),    ("<op>")) \

#define X6(a, b, c, d) #a,
char *typestring[] = { OBJECT_TYPES };
#undef X6

// the Object type
//forward reference so s_object can contain s_objects
typedef struct s_object Object;

// the s_object structure:
// a bit convoluted, but it boils down to four members:
// type, flags, length, and payload (union of type-specific data)
// the first named union member is integer, so a simple literal object
// can be created on the fly:
// Object o = {integertype,0,0,4028}; //create an int object, value: 4028
// Object nl = {nulltype,0,0,0};
struct s_object {
#define X6(a, b, c, d) a,
    enum e_type { OBJECT_TYPES } type;
#undef X6
unsigned int flags;
#define Fread  1
#define Fwrite 2
#define Fexec  4
#define Fxflag 8
size_t length; //for lint, was: unsigned int
#define X6(a, b, c, d) c;
    union { OBJECT_TYPES };
#undef X6
};
```

One big problem was the printf format strings. While it looks cool, it's just hocus pocus. Since it's only used in one function, overuse of the macro actually separated information that should be together; and it makes the function unreadable by itself. The obfuscation is doubly unfortunate in a debugging function like this one.

```
//print the object using the type's format specifier from the macro
//used by O_equal (ps: =) and O_equalequal (ps: ==)
void printobject(Object o) {
    switch (o.type) {
#define X6(a, b, c, d) \
        case a: printf d; break;
OBJECT_TYPES
#undef X6
    }
}
```

So don't get carried away. Like I did.

Share Improve this answer Follow edited Jul 26, 2013 at 9:20 answered Jul 9, 2011 at 18:15

luser droog
19.1k 3 54 105

1 I've been looking into a few different libraries to deal with "objects" in C - like Cello and GObject but they both took it a bit far for my taste.. This post and your Github code on the other hand - great stuff, thanks for the inspiration. :) – Christoffer Bubach Jul 30, 2020 at 2:24

That's very nice to hear. I studied those, too, as well as looking at the Lisp 1.1 manual. The most recent set of objects I've made is for parser combinators. I got the GC really small and simple there. Be sure to let me know what you're building. This kind of stuff always seems to result in something cool. :) – luser droog Jul 30, 2020 at 8:11

9

Some real-world uses of X-Macros by popular and large projects:

Java HotSpot

In the Oracle HotSpot Virtual Machine for the Java® Programming Language, there is the file `globals.hpp`, which uses the `RUNTIME_FLAGS` in that way.

See the source code:

- JDK 7

- JDK 8
- JDK 9

Chromium

The list of network errors in net_error_list.h is a long, long list of macro expansions of this form:

```
NET_ERROR(IO_PENDING, -1)
```

It is used by net_errors.h from the same directory:

```
enum Error {
  OK = 0,

#define NET_ERROR(label, value) ERR_ ## label = value,
#include "net/base/net_error_list.h"
#undef NET_ERROR
};
```

The result of this preprocessor magic is:

```
enum Error {
  OK = 0,
  ERR_IO_PENDING = -1,
};
```

What I don't like about this particular use is that the name of the constant is created dynamically by adding the `ERR_`. In this example, `NET_ERROR(IO_PENDING, -100)` defines the constant `ERR_IO_PENDING`.

Using a simple text search for `ERR_IO_PENDING`, it is not possible to see where this constant it defined. Instead, to find the definition, one has to search for `IO_PENDING`. This makes the code hard to navigate and therefore adds to the obfuscation of the whole code base.

Share Improve this answer Follow

edited Jan 25, 2020 at 18:02

answered Jul 9, 2011 at 16:28
Roland Illig
40.8k 11 88 122

Ref: hg.openjdk.java.net/jdk8/jdk8/hotspot/file/tip/src/share/vm/... – kevinarpe Dec 12, 2015 at 13:26

Could you include some of that code? This is effectively a link-only answer as it currently stands.
– TankorSmash Nov 1, 2017 at 0:27

6

I use a pretty massive X-macro to load contents of INI-file into a configuration struct, amongst other things revolving around that struct.

This is what my "configuration.def" -file looks like:

```
#define NMB_DUMMY(...) X(__VA_ARGS__)
#define NMB_INT_DEFS \
   TEXT("long int") , long , , , GetLongValue , _ttol , NMB_SECT , SetLongValue ,

#define NMB_STR_DEFS NMB_STR_DEFS__(TEXT("string"))
#define NMB_PATH_DEFS NMB_STR_DEFS__(TEXT("path"))

#define NMB_STR_DEFS__(ATYPE) \
  ATYPE ,  basic_string<TCHAR>* , new basic_string<TCHAR>\
  , delete , GetValue , , NMB_SECT , SetValue , *

/* X-macro starts here */

#define NMB_SECT "server"
NMB_DUMMY(ip,TEXT("Slave IP."),TEXT("10.11.180.102"),NMB_STR_DEFS)
NMB_DUMMY(port,TEXT("Slave portti."),TEXT("502"),NMB_STR_DEFS)
NMB_DUMMY(slaveid,TEXT("Slave protocol ID."),0xff,NMB_INT_DEFS)
.
. /* And so on for about 40 items. */
```

It's a bit confusing, I admit. It quickly become clear that I don't actually want to write all those type declarations after every field-macro. (Don't worry, there's a big comment to explain everything which I omitted for brevity.)

And this is how I declare the configuration struct:

```
typedef struct {
#define X(ID,DESC,DEFVAL,ATYPE,TYPE,...) TYPE ID;
#include "configuration.def"
#undef X
  basic_string<TCHAR>* ini_path;  //Where all the other stuff gets read.
  long verbosity;                 //Used only by console writing functions.
} Config;
```

Then, in the code, firstly the default values are read into the configuration struct:

```
#define
X(ID,DESC,DEFVAL,ATYPE,TYPE,CONSTRUCTOR,DESTRUCTOR,GETTER,STRCONV,SECT,SETTER,...)
\
  conf->ID = CONSTRUCTOR(DEFVAL);
#include "configuration.def"
#undef X
```

Then, the INI is read into the configuration struct as follows, using library SimpleIni:

```
#define
X(ID,DESC,DEFVAL,ATYPE,TYPE,CONSTRUCTOR,DESTRUCTOR,GETTER,STRCONV,SECT,SETTER,DEREF.
  DESTRUCTOR (conf->ID);\
  conf->ID  = CONSTRUCTOR( ini.GETTER(TEXT(SECT),TEXT(#ID),DEFVAL,FALSE) );\
  LOG3A(<< left << setw(13) << TEXT(#ID) << TEXT(": ")  << left << setw(30)\
    << DEREF conf->ID << TEXT(" (") << DEFVAL << TEXT(").") );
#include "configuration.def"
#undef X
```

And overrides from commandline flags, that also are formatted with the same names (in GNU long form), are applies as follows in the foillowing manner using library SimpleOpt:

```
enum optflags {
#define X(ID,...) ID,
#include "configuration.def"
#undef X
  };
  CSimpleOpt::SOption sopt[] = {
#define X(ID,DESC,DEFVAL,ATYPE,TYPE,...) {ID,TEXT("--") #ID TEXT("="),
SO_REQ_CMB},
#include "configuration.def"
#undef X
    SO_END_OF_OPTIONS
  };
  CSimpleOpt ops(argc,argv,sopt,SO_O_NOERR);
  while(ops.Next()){
    switch(ops.OptionId()){
#define
X(ID,DESC,DEFVAL,ATYPE,TYPE,CONSTRUCTOR,DESTRUCTOR,GETTER,STRCONV,SECT,...) \
  case ID:\
    DESTRUCTOR (conf->ID);\
    conf->ID = STRCONV( CONSTRUCTOR (  ops.OptionArg() ) );\
    LOG3A(<< TEXT("Omitted ")<<left<<setw(13)<<TEXT(#ID)<<TEXT(" : ")<<conf-
>ID<<TEXT(" ."));\
    break;
#include "configuration.def"
#undef X
    }
  }
```

And so on, I also use the same macro to print the --help -flag output and sample default ini file, configuration.def is included 8 times in my program. "Square peg into a round hole", maybe; how would an actually competent programmer proceed with this? Lots and lots of loops and string processing?

Share Improve this answer Follow

answered Jul 17, 2015 at 15:56

VITTUIX-MAN
346 2 8

I like to use X macros for creating 'rich enumerations' which support iterating the enum values as well as getting the string representation for each enum value:

5

```
#define MOUSE_BUTTONS \
X(LeftButton, 1)   \
X(MiddleButton, 2) \
X(RightButton, 4)

struct MouseButton {
  enum Value {
    None = 0
#define X(name, value) ,name = value
MOUSE_BUTTONS
#undef X
  };

  static const int *values() {
    static const int a[] = {
      None,
#define X(name, value) name,
    MOUSE_BUTTONS
#undef X
      -1
    };
    return a;
  }

  static const char *valueAsString( Value v ) {
#define X(name, value) static const char str_##name[] = #name;
MOUSE_BUTTONS
#undef X
    switch ( v ) {
      case None: return "None";
#define X(name, value) case name: return str_##name;
MOUSE_BUTTONS
#undef X
    }
    return 0;
  }
};
```

This not only defines a `MouseButton::Value` enum, it also lets me do things like

```
// Print names of all supported mouse buttons
for ( const int *mb = MouseButton::values(); *mb != -1; ++mb ) {
    std::cout << MouseButton::valueAsString( (MouseButton::Value)*mb ) << "\n";
}
```

Share Improve this answer Follow

answered Feb 20, 2015 at 9:23

Frerich Raabe

91.3k 19 115 208

1

https://github.com/whunmr/DataEx

I am using the following xmacros to generate a C++ class, with serialize and deserialize functionality built in.

```
#define __FIELDS_OF_DataWithNested(_)  \
  _(1, a, int  )                       \
  _(2, x, DataX)                       \
  _(3, b, int  )                       \
  _(4, c, char )                       \
  _(5, d, __array(char, 3))            \
  _(6, e, string)                      \
  _(7, f, bool)

DEF_DATA(DataWithNested);
```

Usage:

```
TEST_F(t, DataWithNested_should_able_to_encode_struct_with_nested_struct) {
    DataWithNested xn;
    xn.a = 0xCAFEBABE;
    xn.x.a = 0x12345678;
    xn.x.b = 0x11223344;
    xn.b = 0xDEADBEEF;
    xn.c = 0x45;
    memcpy(&xn.d, "XYZ", strlen("XYZ"));

    char buf_with_zero[] = {0x11, 0x22, 0x00, 0x00, 0x33};
    xn.e = string(buf_with_zero, sizeof(buf_with_zero));
    xn.f = true;

    __encode(DataWithNested, xn, buf_);

    char expected[] = { 0x01, 0x04, 0x00, 0xBE, 0xBA, 0xFE, 0xCA,
                        0x02, 0x0E, 0x00 /*T and L of nested X*/,
                        0x01, 0x04, 0x00, 0x78, 0x56, 0x34, 0x12,
                        0x02, 0x04, 0x00, 0x44, 0x33, 0x22, 0x11,
                        0x03, 0x04, 0x00, 0xEF, 0xBE, 0xAD, 0xDE,
                        0x04, 0x01, 0x00, 0x45,
                        0x05, 0x03, 0x00, 'X', 'Y', 'Z',
                        0x06, 0x05, 0x00, 0x11, 0x22, 0x00, 0x00, 0x33,
                        0x07, 0x01, 0x00, 0x01};

    EXPECT_TRUE(ArraysMatch(expected, buf_));
}
```

Also, another example is in https://github.com/whunmr/msgrpc.

Share Improve this answer Follow

edited Apr 13, 2019 at 2:59
Peter Mortensen
30.7k 22 106 131

answered Mar 9, 2017 at 9:32
whunmr
2,435 3 22 35

1

Chromium has an interesting variation of a X-macro at dom_code_data.inc. Except it's not just a macro, but an entirely separate file. This file is intended for keyboard input mapping between different platforms' scancodes, USB HID codes, and string-like names.

The file contains code like:

```
DOM_CODE_DECLARATION {

  //            USB     evdev    XKB     Win     Mac   Code
  DOM_CODE(0x000000, 0x0000, 0x0000, 0x0000, 0xffff, NULL, NONE), // Invalid
...
};
```

Each macro invocation actually passes in 7 arguments, and the macro can choose which arguments to use and which to ignore. One usage is to map between OS keycodes and platform-independent scancodes and DOM strings. Different macros are used on different OSes to pick the keycodes appropriate for that OS.

```
// Table of USB codes (equivalent to DomCode values), native scan codes,
// and DOM Level 3 |code| strings.
#if defined(OS_WIN)
#define DOM_CODE(usb, evdev, xkb, win, mac, code, id) \
  { usb, win, code }
#elif defined(OS_LINUX)
#define DOM_CODE(usb, evdev, xkb, win, mac, code, id) \
  { usb, xkb, code }
#elif defined(OS_MACOSX)
#define DOM_CODE(usb, evdev, xkb, win, mac, code, id) \
  { usb, mac, code }
#elif defined(OS_ANDROID)
#define DOM_CODE(usb, evdev, xkb, win, mac, code, id) \
  { usb, evdev, code }
#else
#define DOM_CODE(usb, evdev, xkb, win, mac, code, id) \
  { usb, 0, code }
#endif
#define DOM_CODE_DECLARATION const KeycodeMapEntry usb_keycode_map[] =
#include "ui/events/keycodes/dom/dom_code_data.inc"
#undef DOM_CODE
#undef DOM_CODE_DECLARATION
```

Share Improve this answer Follow

answered Jul 16, 2020 at 14:20

nyanpasu64
2,885 2 24 31

0

My humble example:

One of the steps to speed up the FFmpeg HEVC decoder - hardcode a matrix consisting of only three rows of small integer coefficients, which is used in several places:
https://github.com/aliakseis/FFmpegPlayer/commit/53a28b61cd98e1dda6d04251b713d39122c021d2#diff-8c65aa37510be2621e7b5a550a33c445b4c85607a789c9b483c2e78cdffcd65bL607

Share Improve this answer Follow

answered Feb 20 at 11:13

Aliaks Aliaksei Sanko
11 1 3