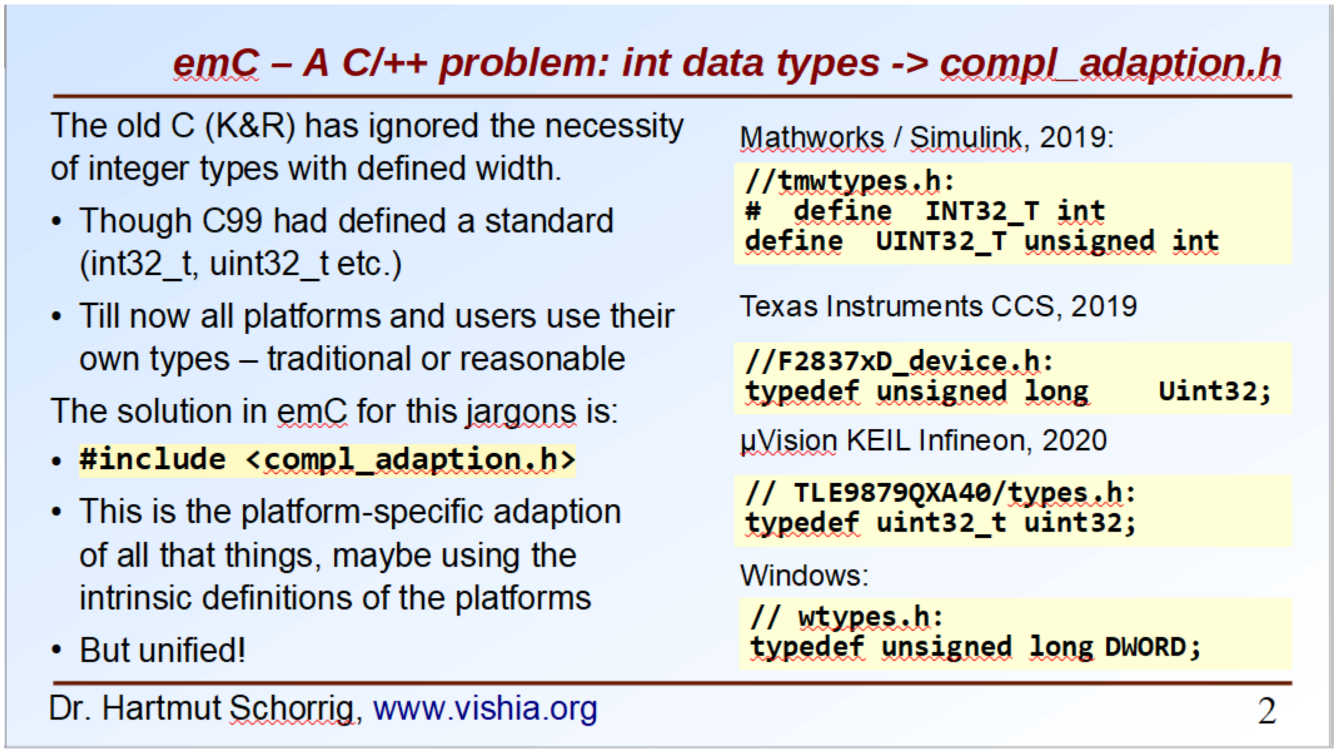
Dr. Hartmut Schorrig, www.vishia.org, 2021-01-21

Embedded multiplatform C/C++ Approach

Table of Contents

1 Necessity of compl_adaption.h



emC – A C/++ problem: int data types -> compl_adaption.h

The old C (K&R) has ignored the necessity of integer types with defined width.

- Though C99 had defined a standard (int32_t, uint32_t etc.)
- Till now all platforms and users use their own types – traditional or reasonable

The solution in emC for this jargons is:

- `#include <compl_adaption.h>`
- This is the platform-specific adaption of all that things, maybe using the intrinsic definitions of the platforms
- But unified!

Mathworks / Simulink, 2019:

```
//tmwtypes.h:
#  define  INT32_T int
define  UINT32_T unsigned int
```

Texas Instruments CCS, 2019

```
//F2837xD_device.h:
typedef unsigned long    Uint32;
```

µVision KEIL Infineon, 2020

```
// TLE9879QXA40/types.h:
typedef uint32_t uint32;
```

Windows:

```
// wtypes.h:
typedef unsigned long DWORD;
```

Dr. Hartmut Schorrig, www.vishia.org 2

As the slide shows the C99 types for bit width fixed integer data types are not present overall. One reason is - the tradition. Often used and familiar type identifier are used furthermore. It is also a problem of legacy code maintenance. The other reason: The standard fix width types in C99 like `int_32_t` etc. are not compiler-intrinsic. They are defined only in a special header file `stdint.h`. Usual this types are defined via `typedef`. This may be disable compatibility. An `int_32_t` is not compatible with a maybe user defined legacy `INT32`. This is complicating. Usage of `stdint.h` is not a sufficient solution. It is too specific and too unflexible.

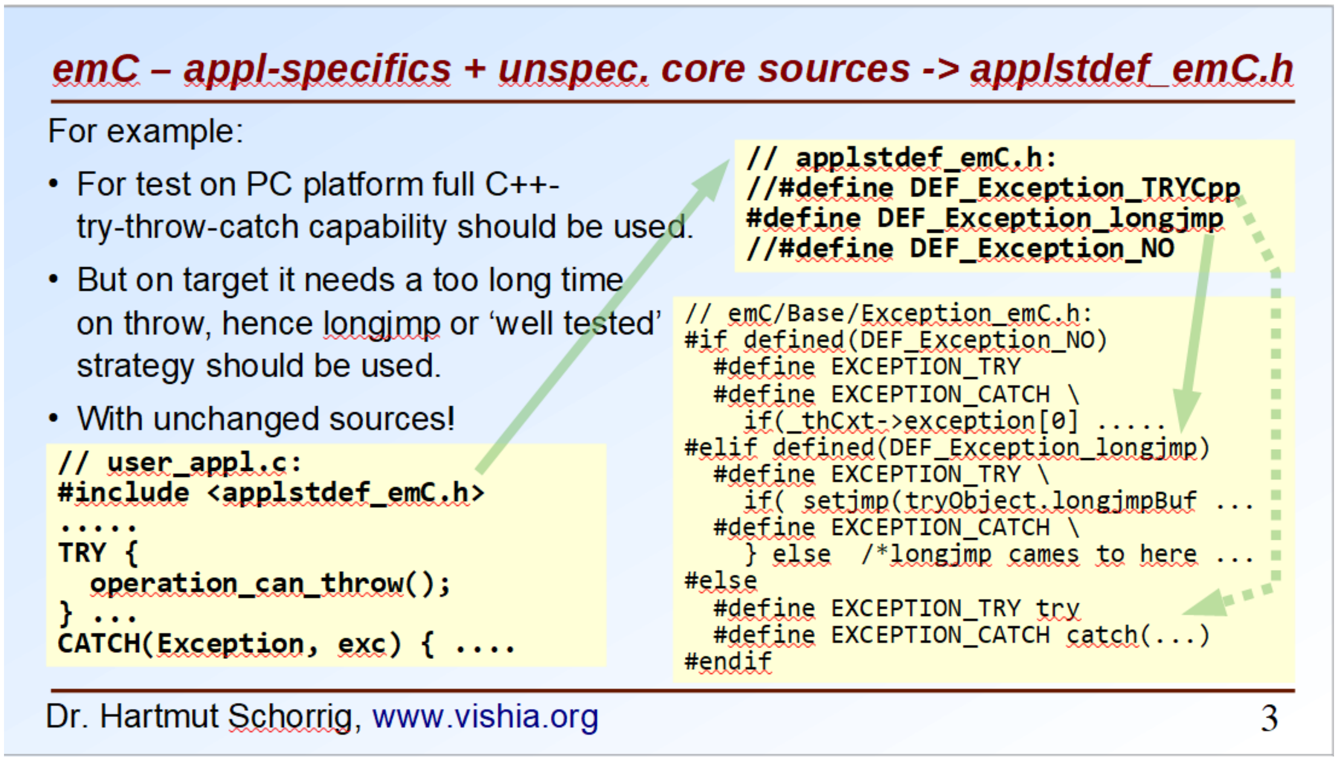
The `compl_adaption.h` should be defined and maintained by the user (not by the compiler tool) or by - the emC library guidelines. It can be enhanced by the user's legacy types in a compatible form. It can include `stdint.h` if it is convenient for the specific platform - or replace this content.

The `compl_adaption.h` should be included in all user's sources, as first one. It should never force a contradiction to other included files, else for specific non changeable system files for example `wintypes.h` which may be necessary only for adaptions of that operation system. Then the contradictions can be resolves via `#undef` of disturbing definitions of the system specific afterwords defined things.

System specific include files such as `wintypes.h` or `windows.h` should ***never*** be included in user's sources which are not especially for the specified system. It should be also true if some definitions should match the expectiations of the user's source independent of the specific system.

The `compl_adaption.h` contains some more usefully definitions, see .../Base/compl_adaption_h.html.

2 What is applstdef_emC.h - necessity for emC



emC – appl-specifics + unspec. core sources -> applstdef_emC.h

For example:

- For test on PC platform full C++-try-throw-catch capability should be used.
- But on target it needs a too long time on throw, hence longjmp or 'well tested' strategy should be used.
- With unchanged sources!

```
// user_appl.c:
#include <applstdef_emC.h>
.....
TRY {
  operation_can_throw();
} ...
CATCH(Exception, exc) { ....
```

```
// applstdef_emC.h:
//#define DEF_Exception_TRYCpp
#define DEF_Exception_longjmp
//#define DEF_Exception_NO
```

```
// emC/Base/Exception_emC.h:
#if defined(DEF_Exception_NO)
  #define EXCEPTION_TRY
  #define EXCEPTION_CATCH \
    if(_thCxt->exception[0] .....
#elif defined(DEF_Exception_longjmp)
  #define EXCEPTION_TRY \
    if( setjmp(tryObject.longjmpBuf ...
  #define EXCEPTION_CATCH \
    } else  /*longjmp cames to here ...
#else
  #define EXCEPTION_TRY try
  #define EXCEPTION_CATCH catch(...)
#endif
```

Dr. Hartmut Schorrig, www.vishia.org 3

The **`applstdef_emC.h`** should be included for all sources, which uses files from the ***emC*** concept. Hence it is not necessary for common driver, only hardware depending, but for user sources. **`applstdef_emC.h`** includes **`compl_adaption.h`**, only one of this file is necessary.

The ***emC*** concept offers some "language extensions" for portable programming (*multiplatform*). That are usual macros, which can be adapted to the platform requirements. For that the **`applstdef_emC.h`** should contain (use a template!) some compiler switches which can be set also platform specific for an application or application specific..

The example shows the selection of an error or exception handling approach. Generally usage of TRY..CATCH or ASSERT is recommended. The user's application should not regard about "how to do that", because often the sources should be reuseable (not really for exact this application), or the implementation on different platforms should use different types of exception handling - without adaption of the sources.

The exception handling and its approaches are presented on Base/ThCxtExc_emC.html .

- Some Variants usage the base class **`ObjectJc`** for Reflection and Types are presented on Base/ObjectJc_en.html. It can be a simple base **`struct`** for poor platforms, or can contain some more information which characterizes all data (basing on **`ObjectJc`**) in a unique way.
- Reflection usage, presented on Base/ClassJc_en.html can be used with elaborately text information for symbolic access to all data, with a "InspcTargetProxy" concept for symbolic access to a poor target system, or only for a maybe simple type test.

See Base/applstdef_emC_h.html

3 HAL - Hardware Adaption Layer - and file arrangement for embedded targets

TODO