CC2511 Week 4: Lecture 1

“ It's not at all important to get it right the first time. It's vitally important to get it right the last time. ”

Andrew Hunt and David Thomas

This Lecture

- Part 1: More on the C language
- Part 2: Code generators

C language: More on the pre-processor

- Recall that we used the #include statement to copy the contents of header files into the current C file.
- Lines beginning with # are ‘directives’ to the pre-processor.
- The pre-processor can also implement logic and transform the code.

Pre-processor macros

- The pre-processor supports logic:
- #define NAME (declare NAME as a pre-processor macro)
- #undef NAME (undefine NAME, i.e. undo an earlier #define)
- #ifdef NAME (if NAME is defined)
- #ifndef NAME (if NAME is not defined)
- #else (optional else clause for #ifdef or #ifndef)
- #endif (required after every #ifdef or #ifndef)

Pre-processor conditions

- Header files can #include other headers.
- It’s possible to have the same file included multiple times.
- A “guard” is commonly used to avoid this: (file name “motors.h”)

```
#ifndef MOTORS_H
#define MOTORS_H
// ...
#endif
```

Pre-processor macros

- A “macro” performs a transformation of the source code before it is compiled.
- The pre-processor will expand macros

```
#define MAX_SPEED 100
if (speed > MAX_SPEED)
      speed = MAX_SPEED;
```

- Here, the symbol “MAX_SPEED” is replaced with 100 by the pre-processor (before the code is compiled).

Macro arguments

- Macros can have arguments

```
#define MAKE_INT(n,v) int n = v;

MAKE_INT(speed, 50);
// expands to:   int speed = 50;
```

Dangers of macros

- Use macros with caution! There are many subtle problems.

```
#define inc(x) x+1


int a = 2*inc(x);
// expands to: int a = 2*x+1;
// The multiplication has the highest precedence!
```

C type qualifiers

- Notice the pattern:

[signed | unsigned] [short | long | long long] [type]

```
signed   short     int // 16 bits
unsigned long      int // 32 bits
unsigned long long int // 64 bits
```

Typecasting

- Typecasting means converting from one data type to another, e.g. from integer to float.
- C syntax for typecasting places the new type name in parentheses:

```
int i = 3;
float f = (float)i;
```

Numeric typecasting

- Numeric typecasting converts from one format to another as best as possible.
- Integer to float may introduce a slight rounding error because not all integers can be represented exactly.
- **Float to integer always rounds down.**

```
float f = 3.67;
int i = (int)f; // rounds down to 3
```

Typecasting pointers

- It’s possible to typecast pointers.
- This sidesteps the compiler’s type system and lets you treat any location in memory as if it were any other data type.
- It does not convert between formats!

Example of pointer typecasting

```
float f = 1.1;
float *fp = &f;
uint32_t *ip = (uint32_t *)fp;
```

- The value of *ip is NOT 1 because the floating point number format is completely different from the integer number format.
- The result is an integer with the exact same sequence of bits as the original float. (So you can see the floating point format)

A void pointer

- A void pointer

```
void *ptr;
```

is a pointer that does not signify the data type that it points to.

- Typically used to refer to data that could be of any format, e.g. data to be transmitted over a communication interface.
- Usually the consumer of the data would know the type and then typecast the pointer.

Type promotion

- Consider:

```
uint16_t a = 10;
uint32_t b = 20;
```

- What is the data type of `a+b` ?
- The smaller integer is automatically converted to the larger size.
- Therefore: `a+b` is a `uint32_t`

Type promotion with floats

- Consider:

```
uint16_t a = 10;
float b = 20.1;
float c = a+b; // the integer is promoted to a float
```

Division

- A trap for the unwary:

```
int a = 1;
int b = 3;

float c = a/b;
```

- After the code is run, c contains zero. Why?

Division

- A trap for the unwary:

```
int a = 1;
int b = 3;

float c = a/b;
```

- After the code is run, c contains zero. Why?
- An integer division is done which always rounds down.

Division

- Floating point division:

```
int a = 1;
int b = 3;

float c = (float)a/b;
```

- At least one of the operands must be cast to float.
- The typecast has a higher precedence than the division.

Floating point operations in embedded systems

- Many microprocessors do not include floating point instructions.
  - The K20D50M board has no floating point unit.
  - The K22F board does have floating point instructions.
- If you use floats or doubles on the K20D50M board, the compiler will insert function calls to code that **emulates floating point in software**.
- This can be quite slow. If high performance is critical, then it's necessary to use integer math (or switch to a CPU with hardware support for floats).

C language syntax

- Finish up the C language reference.

Part 2: Code generators

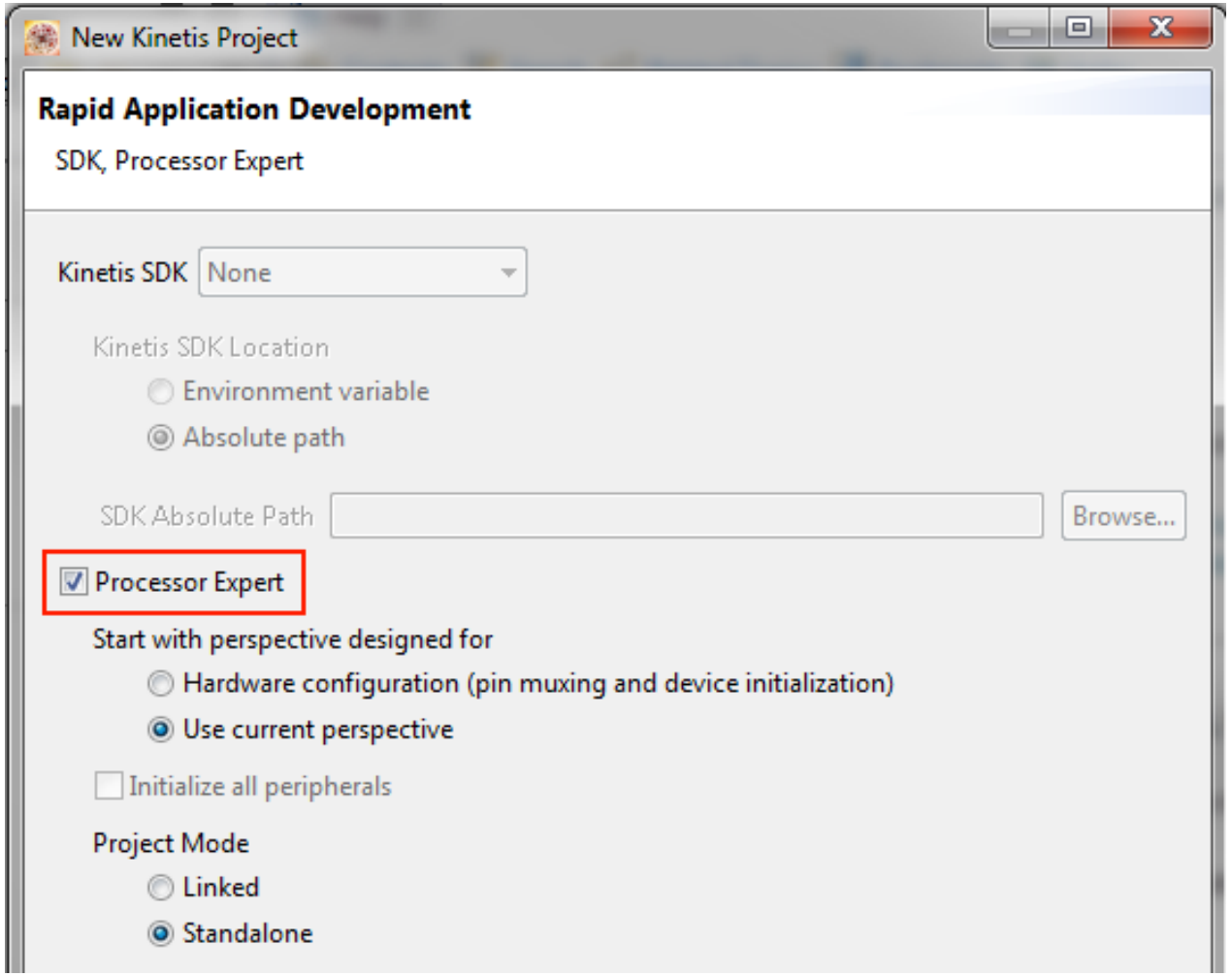
- In last week's lab you set registers to configure GPIO.
- There was a lot of generic code that is:
  - Not fun to write,
  - Prone to silly trivial errors, and
  - Repetitive
- GPIO is the simplest of all peripherals. Imagine trying to configure USB by hand!
- Can all this be automated?

Introduction to Processor Expert

- Processor Expert is a code generator built into Kinetis Design Studio.
- It generates peripheral initialisation code for you.

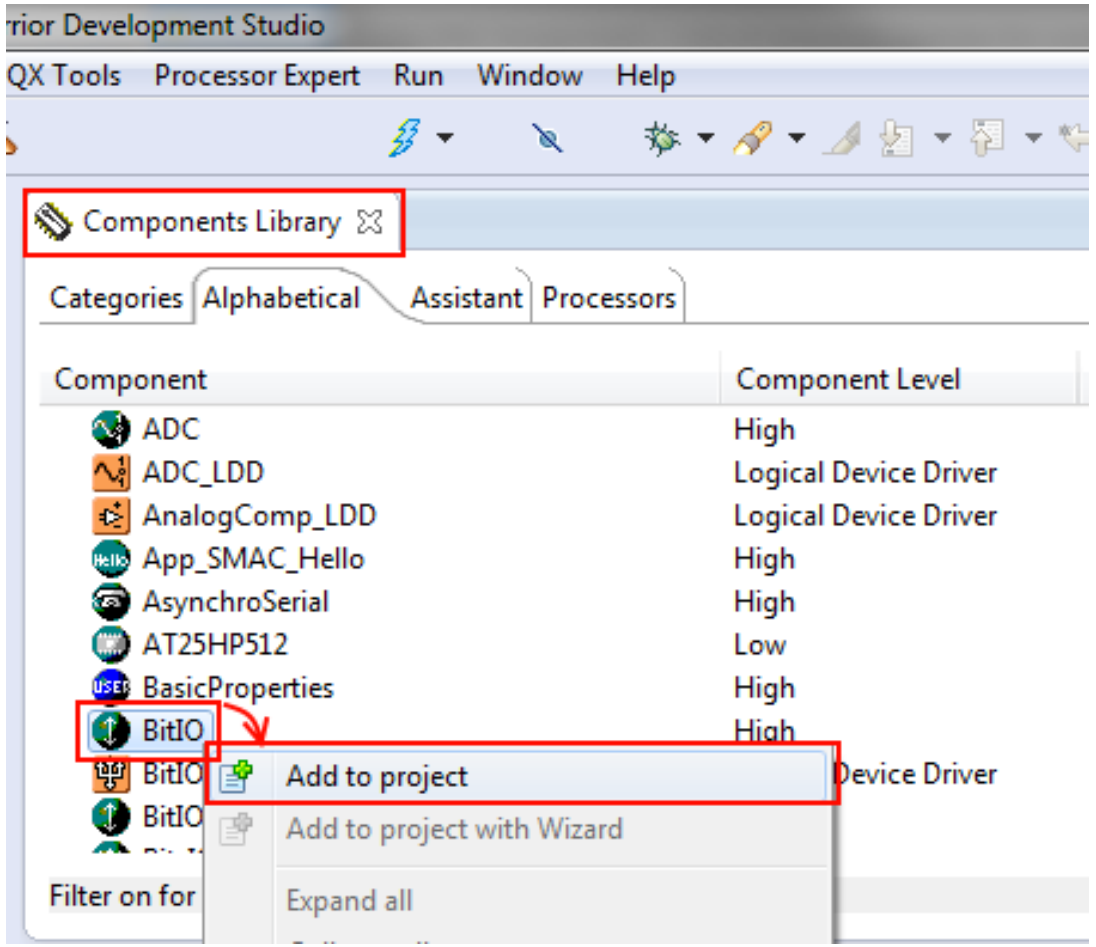
Enabling Processor Expert

- Enable Processor Expert when creating a new project.



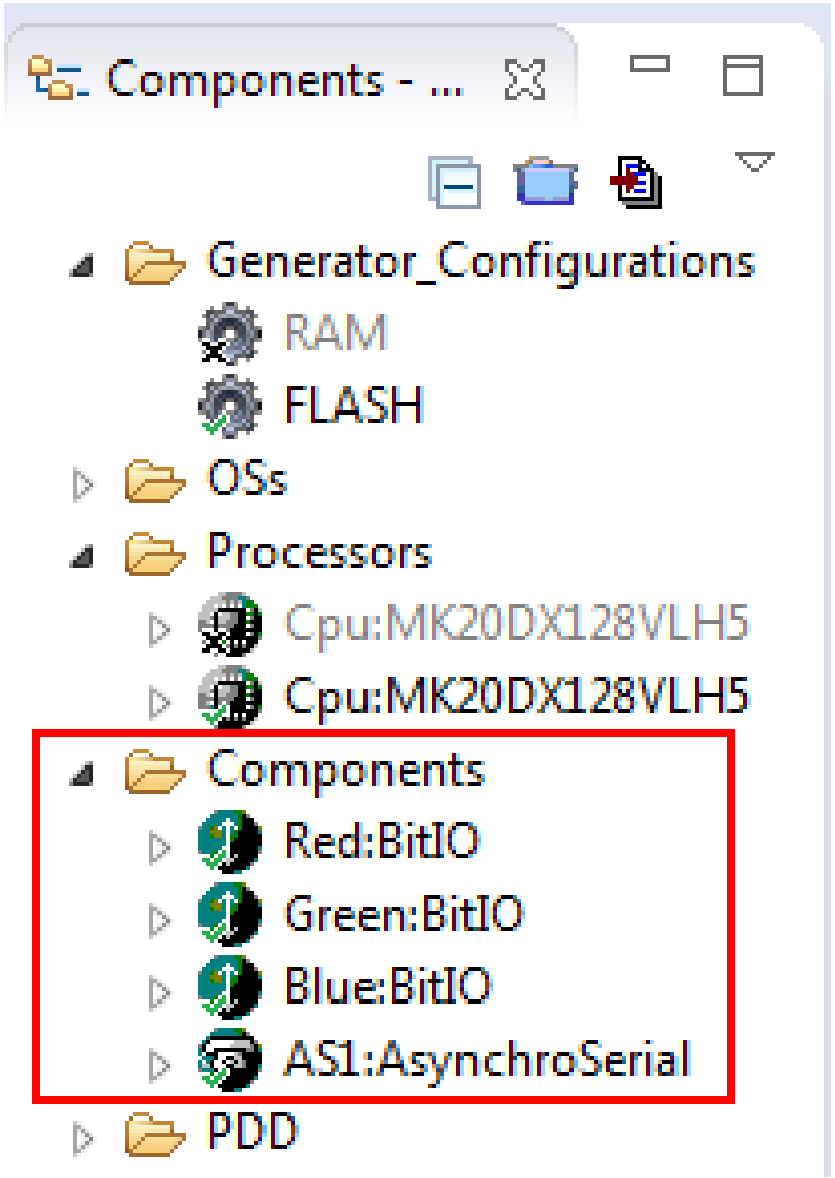
Processor Expert

- There are “**Components**” that generate initialisation code for most peripherals.
- Example: **BitIO** configures a single bit for GPIO.
- Right click on a component and choose “Add to project”.



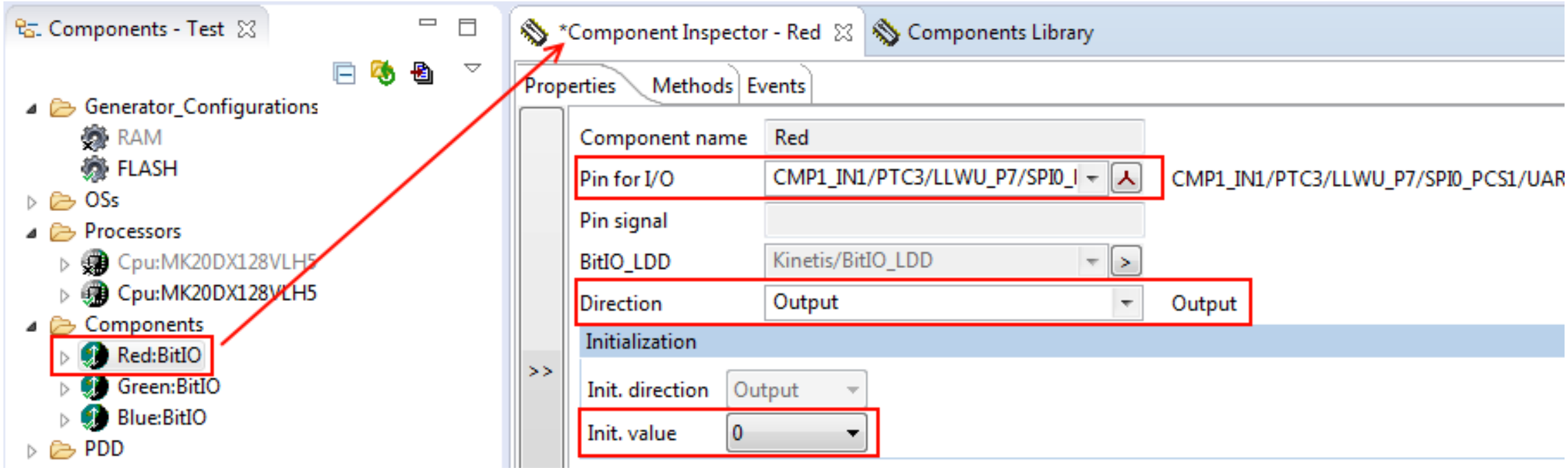
Components view

- The components added to a project appears in the “Components” tab.
- Example: This project includes a three BitIO components: “Red”, “Green”, and “Blue”



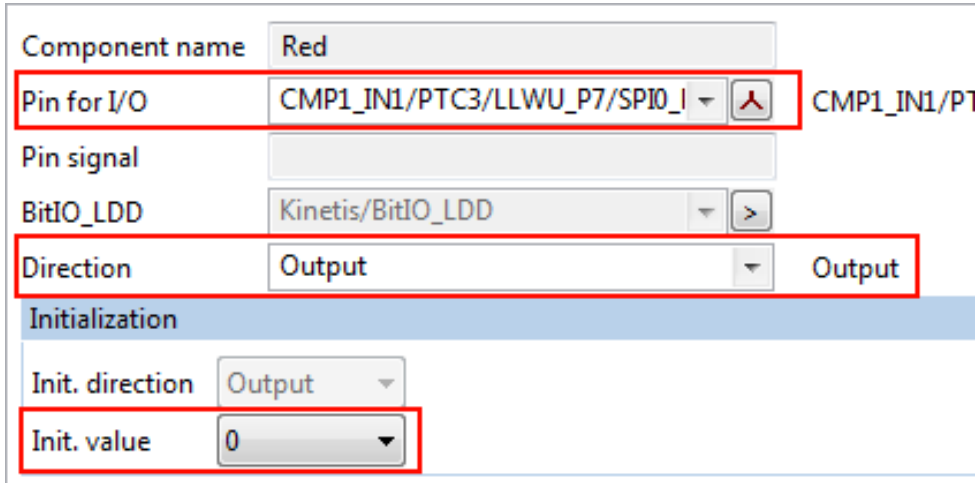
Example: Configuring BitIO

- Click on the component to view its properties in the Component Inspector

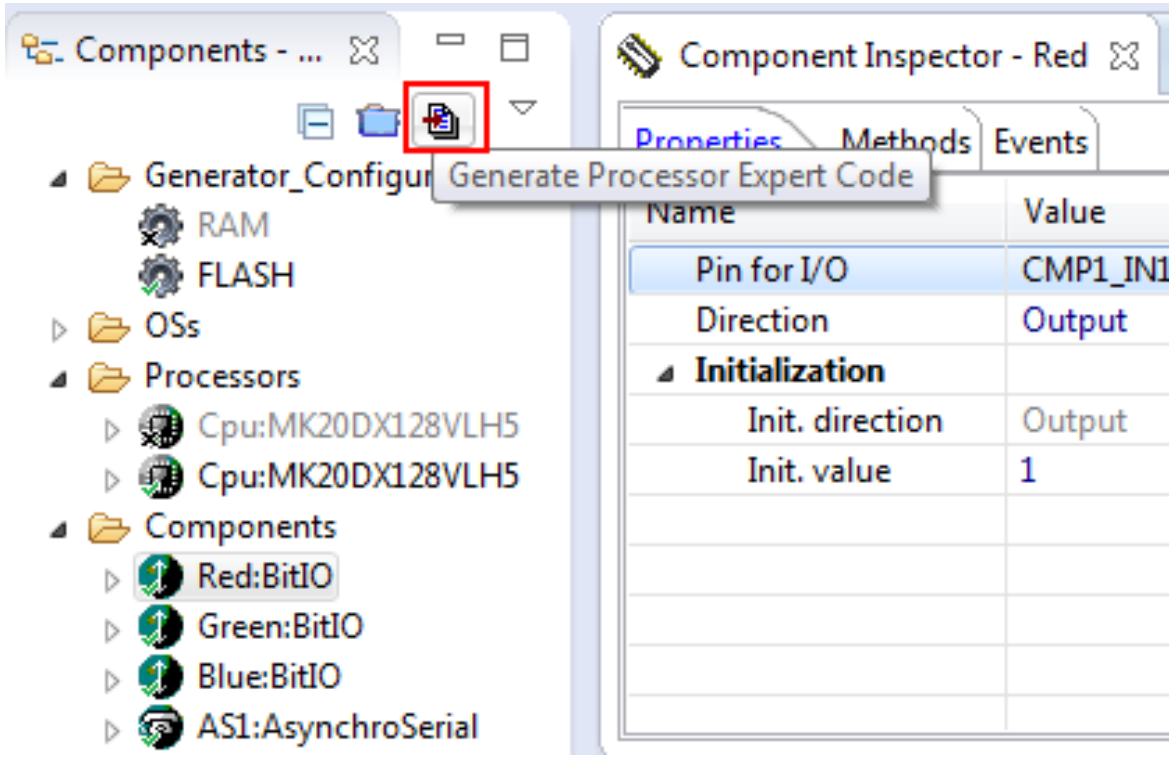


Example: Configuring BitIO

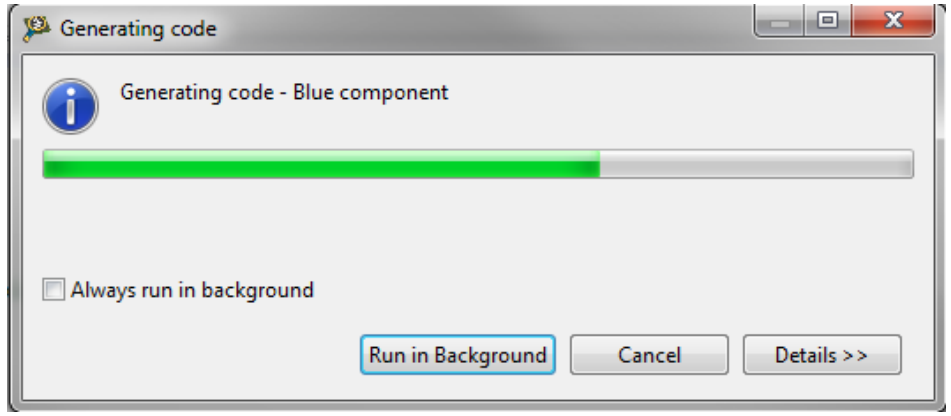
- Set the direction (here, **Output**)
- Set the pin (here, **PTC3**)
- Set the initial value (here, **0**)



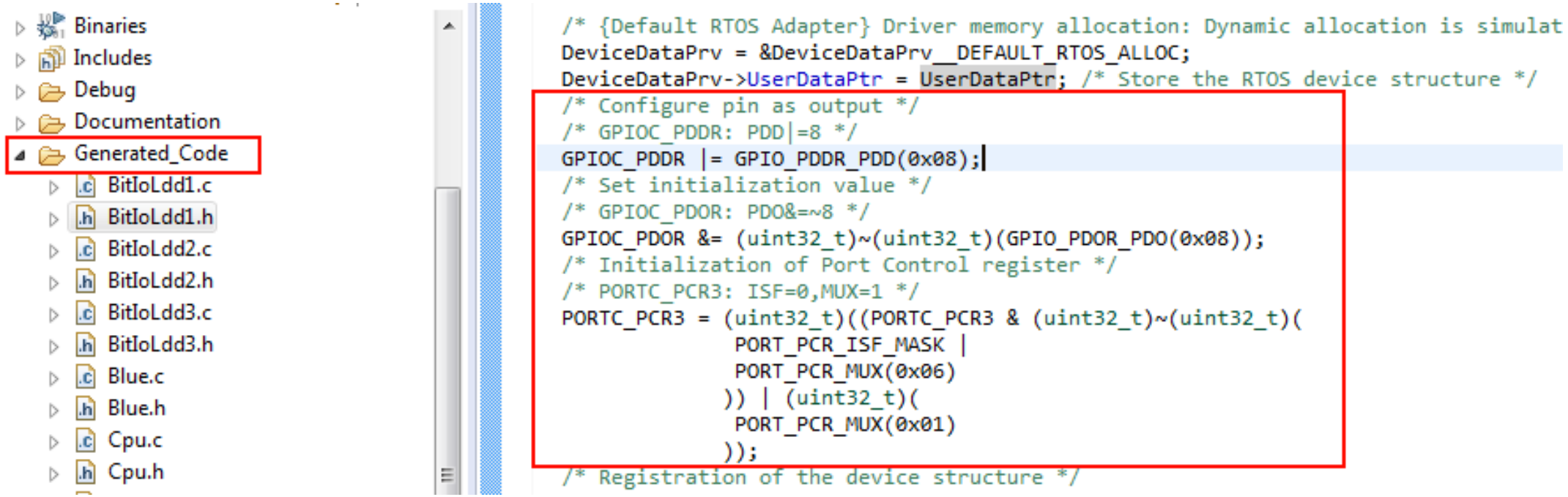
Generating code



- Click the button in the Components view to run the code generator
- **Note: Processor Expert does not automatically regenerate code! If you change the settings, click the button.**

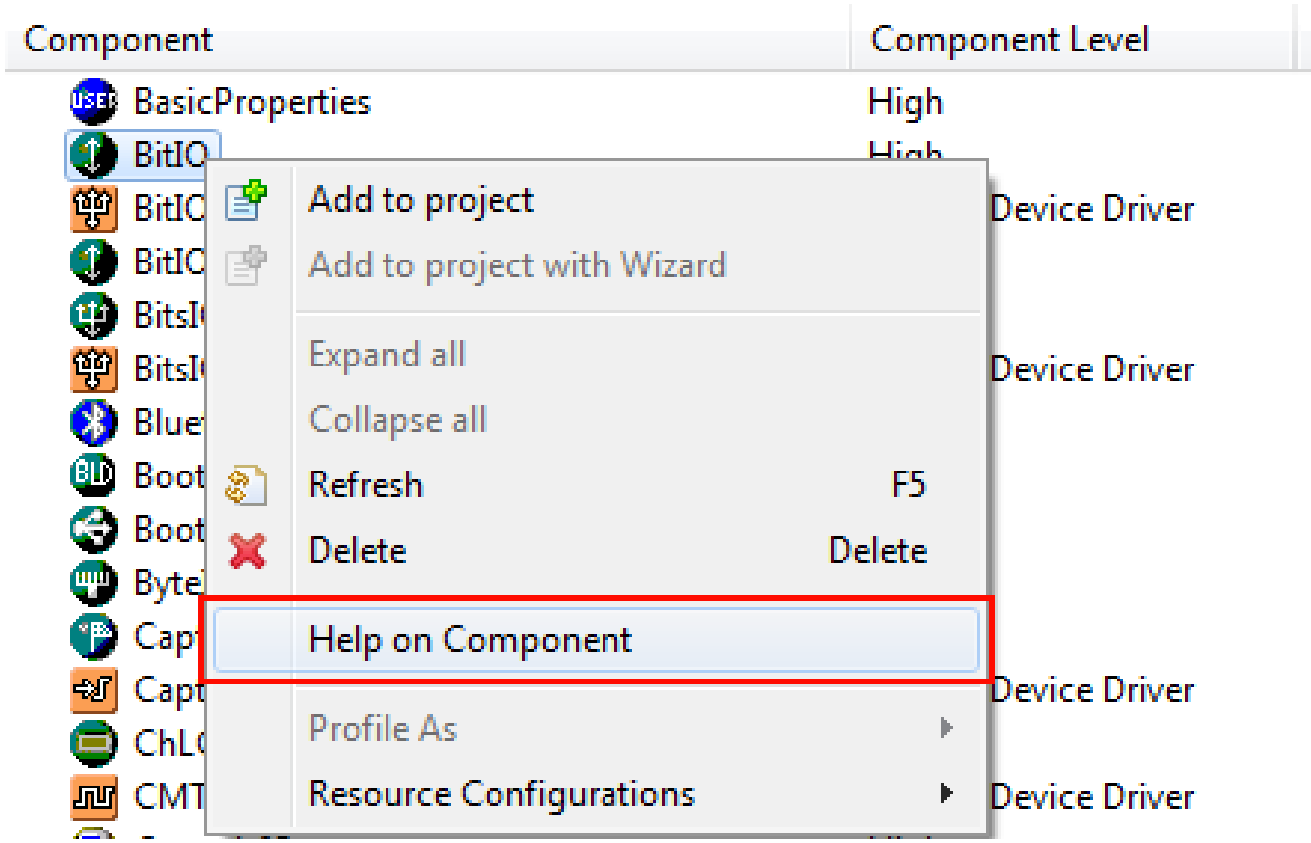


Viewing the generated code



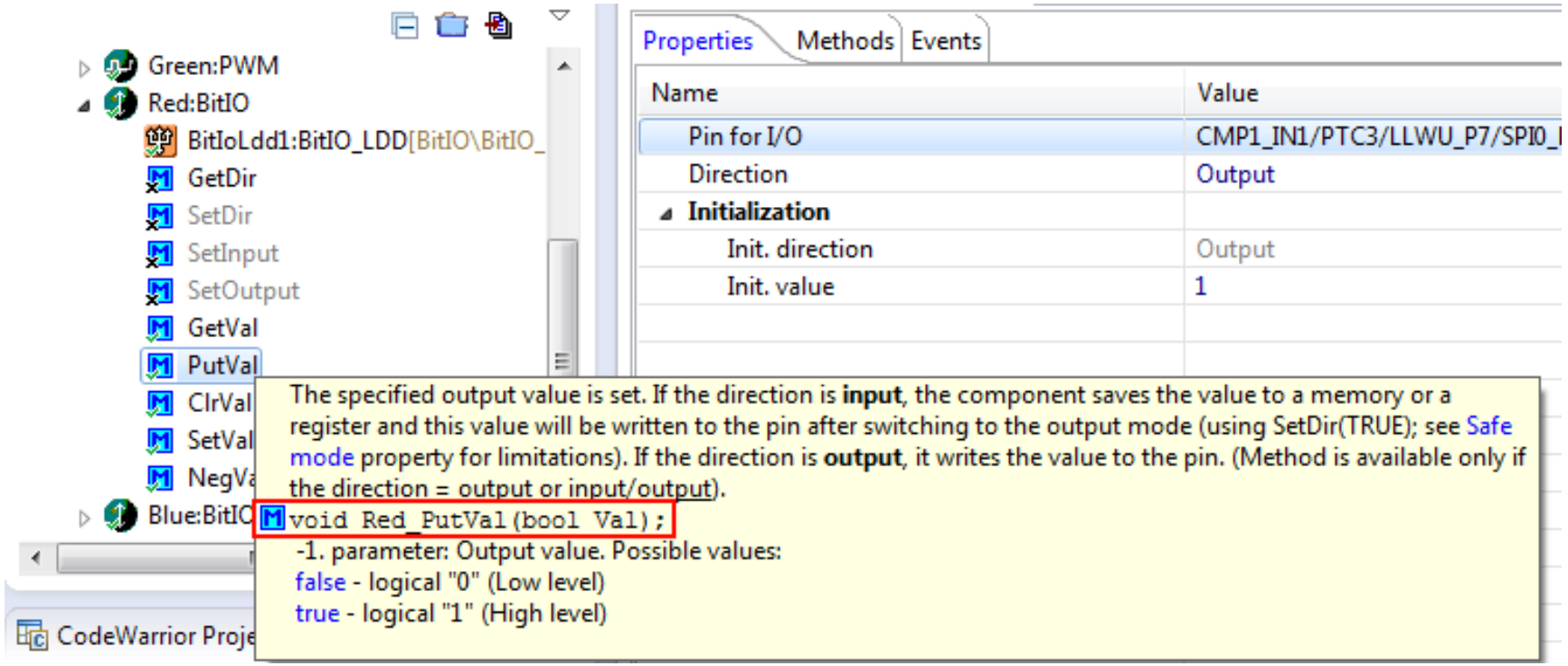
Documentation

- Right click on any component and click **Help on Component** to read the documentation



Calling generated functions

- Function prototypes are visible by resting the mouse cursor over the function name.



Summary

- You’re welcome to use Processor Expert in labs and assignments.
- Processor Expert components don’t always make every feature available to you. Sometimes you still need to manually set registers.