I tried local static buffere, but it still freezes, even as global static buffer ... it seems its only some quirk of the QL, maybe runtime/library ... and there is other weird thing, when I not introduce "printn" commands here the result of this calculation is bad, when they are there, result is ok:

```
void mul_TS32(TS64X* result, TS32 a, TS32 b) {
   TU32 a_unsigned = (TU32)a;
   TU32 b_unsigned = (TU32)b;
   TU64X mul_unsigned;

   //int tmp1 = 0;
   println("mul_TU32 before");

//    print("a:  "); println(itoh(a,8));
//    print("b:  "); println(itoh(b,8));
   //print("au: "); println(itoh(a_unsigned,8));
   //print("bu: "); println(itoh(b_unsigned,8));

   //print("a:  "); println(itoh(a,8));
   //print("au: "); println(itoh(a_unsigned,8));
   //print("b:  "); println(itoh(b,8));
   //print("bu: "); println(itoh(b_unsigned,8));

   mul_TU32(&mul_unsigned, a_unsigned, b_unsigned);
   result->low = mul_unsigned.low;
   result->high = mul_unsigned.high;

   //tmp1 = 0;
   println("mul_TU32 after");

   // Adjust the sign
   if (a < 0) {
      result->high -= b_unsigned;
   }
   if (b < 0) {
      result->high -= a_unsigned;
   }
}
```

It seems like you're encountering two issues with your code:

1. The program freezes when using local static buffers.
2. The output of the multiplication is incorrect when the **`printn`** statements are removed.

For the first issue, it might be helpful to know more about the context of your code, especially how the **`mul_TS32`** function is being used, and what the **`println`** function looks like. It's possible that you're encountering a race condition or memory issue that's causing your program to freeze.

For the second issue, it's possible that the inclusion of **`println`** statements is affecting the behavior of your program. This could be due to compiler optimizations or memory access issues. To help diagnose this issue, try the following steps:

1. Ensure that the **`println`** function is implemented correctly and does not have any side effects. If the **`println`** function is modifying memory or using global/static variables, it could be affecting the behavior of your program.
2. Add memory barriers or volatile qualifiers to your code. This can help prevent the compiler from making certain optimizations that could cause issues with the order of your code execution.
   Example:

   c   Copy code

   ```c
   volatile TU32 a_unsigned = (TU32)a; volatile TU32 b_unsigned = (TU32)b;
   volatile TU64X mul_unsigned;
   ```

3. Try disabling compiler optimizations, if possible. This can help you determine if the issue is related to the optimizations being applied to your code. If the problem goes away with optimizations disabled, you can then focus on identifying which optimization is causing the issue.
4. Use a debugger to step through your code and inspect the values of the variables during the execution. This can help you pinpoint where the values are being corrupted or where unexpected behavior is occurring.

Remember that without the complete context of your code, it might be difficult to provide a definitive solution. However, these steps should help you diagnose and potentially resolve the issue.

how to direct gcc to switch off optimizations?

You can tell GCC to disable optimizations by using the **`-O0`** (capital O followed by zero) flag when compiling your code. Here's an example of how to compile a C source file named **`example.c`** with optimizations disabled:

sh Copy code

```sh
gcc -O0 example.c -o example
```

This command will compile the **`example.c`** file with optimizations disabled and produce an executable named **`example`**. The **`-O0`** flag instructs GCC to prioritize compilation speed and debugging information over execution speed and code size, effectively disabling optimizations.

well, cant be here TS64X defined badly??

void print(const TCHAR* str);
void println(const TCHAR* str);
TCHAR* itoh(TU32 num, TU8 pos);

typedef struct {
    TU32 low;
    TU32 high;
} TU64X;

typedef struct {
    TU32 low;
    TS32 high;

```
} TS64X;


// Function to multiply two unsigned 32-bit integers and return a 64-bit result
void mul_TU32(TU64X* result, TU32 a, TU32 b) {
    TU32 a_low = a & 0xFFFF;
    TU32 a_high = a >> 16;
    TU32 b_low = b & 0xFFFF;
    TU32 b_high = b >> 16;

    TU32 low_low = a_low * b_low;
    TU32 low_high = a_low * b_high;
    TU32 high_low = a_high * b_low;
    TU32 high_high = a_high * b_high;

    TU32 mid1;
    TU32 mid2;

    result->low = low_low;
    result->high = high_high;

    mid1 = low_high << 16;
    mid2 = high_low << 16;

    result->low += mid1;
    if (result->low < mid1) {
        result->high++;
    }

    result->low += mid2;
    if (result->low < mid2) {
        result->high++;
    }

    result->high += low_high >> 16;
    result->high += high_low >> 16;
}

// Function to multiply two signed 32-bit integers and return a 64-bit result
void mul_TS32(TS64X* result, TS32 a, TS32 b) {
```

```
    TU32 a_unsigned = (TU32)a;
    TU32 b_unsigned = (TU32)b;
    TU64X mul_unsigned;

    if (a<0 || b<0) println("mul_TU32 before");

    //print("a:  "); println(itoh(a,8));
    //print("au: "); println(itoh(a_unsigned,8));
    //print("b:  "); println(itoh(b,8));
    //print("bu: "); println(itoh(b_unsigned,8));

    mul_TU32(&mul_unsigned, a_unsigned, b_unsigned);
    result->low = mul_unsigned.low;
    result->high = mul_unsigned.high;

    if (a<0 || b<0) println("mul_TU32 after");

    // Adjust the sign
    if (a < 0) {
        result->high -= b_unsigned;
    }
    if (b < 0) {
        result->high -= a_unsigned;
    }
}

TS32 mul(TS32 a, TS32 b) {
    TS64X result;
    mul_TS32(&result, a, b);
    return (TS32)result.low;
}

TS32 mulh(TS32 a, TS32 b) {
    TS64X result;
    mul_TS32(&result, a, b);
    return result.high;
}

TU32 mulhu(TU32 a, TU32 b) {
    TU64X result;
```

```
    mul_TU32(&result, a, b);
    return result.high;
}

TS32 mulhsu(TS32 a, TU32 b) {
    TS64X result;
    mul_TS32(&result, a, (TS32)b);
    return result. High;
}
```

From the provided code, the **`TS64X`** and **`TU64X`** structures seem to be defined correctly. The issue with the freezing and incorrect results seems to be elsewhere in the code.

Upon closer inspection, I noticed that you are using the **`itoh`** function to convert integers to hexadecimal strings. This function may be causing the issue if it is not implemented correctly or if it is affecting memory in unintended ways.

As a first step, try removing the **`itoh`** function calls in your code to see if it resolves the freezing and incorrect result issues. If the issue persists, try running the code without the **`print`** and **`println`** statements to see if it still reproduces the problem.

In addition, the **`mul`** and **`mulh`** functions are returning the **`low`** and **`high`** parts of the **`TS64X`** structure, respectively. Make sure this is the intended behavior, as these values might not always represent the correct signed multiplication results.

well itoh is never called here, and only on QL results are affected by those lines with println
...

isn't this weird? the unsigned part of TS64X??

```
typedef struct {
    TU32 low;
    TS32 high;
} TS64X;
```

My apologies for the confusion about the **`itoh`** function; you're right that it isn't called in the code you provided.

Regarding the **`TS64X`** structure, you are correct that it might seem a bit odd to have an unsigned low part and a signed high part. However, this design choice could be related to the specific use case or requirements of your code. If you want to have a consistent signed-ness in the **`TS64X`** structure, you can change the **`low`** part to **`TS32`**. But, if the code is working as expected and providing correct results, it might not be necessary to change the structure.

As for the issue with the QL platform, it could be related to platform-specific implementation details, compiler optimizations, or hardware quirks. Since it's difficult to provide a definitive solution without access to the complete context of your code and the platform, I would suggest testing the code on a different platform or using a different compiler to see if the problem persists. If the problem is specific to the QL platform, you might need to look for QL-specific documentation, forums, or resources to find a solution.

---

is this true for gcc for M68k on QL (68008) ??

typedef unsigned long   TU32;
typedef signed long   TS32;
typedef unsigned short TU16;
typedef signed short   TS16;
typedef unsigned char   TU8;
typedef signed char   TS8;

Yes, these typedefs should be correct for the Motorola 68k family, including the 68008 processor used in the Sinclair QL, assuming you are using a standard GCC compiler targeting the m68k architecture. The Motorola 68k family has a 32-bit architecture, which means that **`long`** should be 32 bits wide, **`short`** should be 16 bits wide, and **`char`** should be 8 bits wide.

Here's a summary of the data types and their bit widths for the m68k architecture:

- **`unsigned long`** (TU32) and **`signed long`** (TS32): 32 bits
- **`unsigned short`** (TU16) and **`signed short`** (TS16): 16 bits
- **`unsigned char`** (TU8) and **`signed char`** (TS8): 8 bits

Using these typedefs, you can create platform-specific type aliases for your code that are easier to understand and maintain.

---

if it is in fact 16/32bit CPU isnt here long 64bit? I provided those typedefs to mimic missing stdint on z88dk compiler for Z80 in fact .....

My apologies for the confusion earlier. The Motorola 68k family, including the 68008, is indeed a 16/32-bit CPU architecture, which means that the size of the standard C types might be different from more modern 32-bit or 64-bit architectures.

For the Motorola 68k family, here are the typical sizes for the standard C types:

- **`char`**: 8 bits
- **`short`**: 16 bits
- **`int`**: 16 bits
- **`long`**: 32 bits
- **`long long`**: 64 bits (not available in all compilers)

So, the typedefs you provided should be correct for the 68k architecture, as the **`long`** type is indeed 32 bits wide. However, if you are looking for a portable way to define fixed-width integer types, you can use the **`<stdint.h>`** header, which is available in C99 and later.

c Copy code

```c
#include <stdint.h> typedef uint32_t TU32; typedef int32_t TS32; typedef uint16_t TU16; typedef int16_t TS16; typedef uint8_t TU8; typedef int8_t TS8;
```

If the **`<stdint.h>`** header is not available for your specific compiler or target platform, the typedefs you provided are a reasonable alternative.

well, gcc for M68k dost support stdint ....

If the GCC for M68k supports the `<stdint.h>` header, you can use the standard fixed-width integer types provided by the header in your code, which ensures portability and consistency across different platforms and compilers.

Here's how you can include the `<stdint.h>` header and use the fixed-width integer types:

c

Copy code

```c
#include <stdint.h> // Now you can use the standard fixed-width integer types directly in your code uint32_t a; int32_t b; uint16_t c; int16_t d; uint8_t e; int8_t f;
```

By using the fixed-width integer types from the `<stdint.h>` header, you don't need to create your own typedefs for the integer types. The standard fixed-width integer types provided by `<stdint.h>` will have the same bit width across different platforms and compilers, ensuring that your code is portable and easier to maintain.

well, gcc for M68k doesnt support stdint ....

I apologize for the misunderstanding earlier. If the GCC for M68k doesn't support the `<stdint.h>` header, then you can continue using the typedefs you provided to define fixed-width integer types. Here are the typedefs you provided for reference:

c

Copy code

```c
typedef unsigned long TU32; typedef signed long TS32; typedef unsigned short TU16; typedef signed short TS16; typedef unsigned char TU8; typedef signed char
```

```
TS8;
```

These typedefs should work correctly for the M68k architecture, as the `long` type is 32 bits wide, the `short` type is 16 bits wide, and the `char` type is 8 bits wide. By using these typedefs, you can create platform-specific type aliases for your code that are easier to understand and maintain.

supports C functions overloading by types?

C does not support function overloading by types. Function overloading is a feature of C++ and some other programming languages, where you can have multiple functions with the same name but different parameter types. The correct function to call is determined by the types of the arguments passed at compile time.

In C, you can achieve a similar effect using separate function names for different types or using variadic functions with different types of arguments. However, neither of these options provides true function overloading.

Here's an example of using separate function names for different types:

c

Copy code

```c
int add_int(int a, int b) { return a + b; } float add_float(float a, float b) {
return a + b; } double add_double(double a, double b) { return a + b; }
```

Another option is to use function pointers and create a generic function that can handle different types based on a type identifier. However, this approach can become complex and may not be as efficient or easy to maintain as using separate function names for different types.

In summary, C does not support function overloading by types, but you can achieve similar functionality by using separate function names or by using other techniques such as variadic functions or function pointers.