

**USING TESTED PRINCIPLES AND DESIGN METHODOLOGIES, YOU CAN CREATE RELIABLE, EASY-TO-INTEGRATE, EASY-TO-SUPPORT, AND HIGH-QUALITY CODE WITH A LONG LIFE CYCLE. THE RESULTING C CODE IS PLATFORM-INDEPENDENT AND REUSABLE WITH STANDARDIZED INTERFACES.**

# Tools of the trade: successful DSP-software development and testing

**W**HEN DEVELOPING DSP APPLICATIONS, software developers can encounter a number of typical design obstacles. A poorly organized development process can create many problems, some of which pop up immediately and some of which become apparent much later in the design cycle. To avoid these common complications—from difficult-to-read code to poor interface compatibility—and to streamline the software-development process, you should follow certain rules and methodologies. The following framework enables you to create reliable, easy-to-integrate, and easy-to-support code and to establish an efficient work-sharing mechanism for a group of developers, thus decreasing time to market. This framework is not a universal guide, but it is useful for both novice and experienced developers.

First, review the methodology, and then consider some specific rules for writing DSP software.

## DEVELOPMENT METHODOLOGY

The diagram in **Figure 1** illustrates the DSP-software-development and -testing process. The software-development and -testing process includes the following steps:

### 1. **Developing and simulating core algorithms.**

Using high-level tools and languages, such as C++, Matlab, MathCAD, LabView, and SystemView, is usually more convenient to develop and simulate core algorithms. These tools allow developers to prove the algorithm's main idea and to roughly outline the smaller independent algorithms and submodules that you need to develop.

### 2. **Developing C code for each small algorithm and submodule.**

You need to express every algo-

rithm and submodule of the system in C code to create platform-independent, maintainable, and flexible code and to be able to simulate the code on a PC before porting to a DSP.

**3. Developing test environment and integrating and simulating environment.** An integration and simulation environment enables the integration of small algorithms and submodules. You should create such an environment for the whole system to simulate it on a PC. This environment usually provides a rich set of visualization tools that allow you to observe the behavior of the system as a whole and of each separate algorithm. To enable the development and execution of different test cases with a set of different input test vectors, you should also create a test environment, which may be a part of the simulation environment. You should test this environment on its own to ensure that it completely covers all possible algorithm states. You can perform this step in parallel with the second step.

**4. Independent testing of each small algorithm or submodule on a PC.** For each submodule, you can create a stand-alone test environment to test the submodule before integration with the whole system.

### 5. **Integrating small algorithms and submodules using integration and simulation environment.**

You should test these elements together as a whole system using the PC test environment. In addition to stand-alone algorithms, you should simulate and test the system that they run in on a PC before porting to a DSP microprocessor. The simulation should cover as many configurations and states of the system as possible. You should achieve

all the required characteristics of the algorithm/system at this stage before porting to DSP hardware. If you cannot achieve some characteristics or find problems during simulation, development should return to the second step to introduce corrections to C code. Sometimes, you need to return to the first step.

**6. Converting critical functions to assembly of DSP.** Usually, converting the whole code to assembly code is unnecessary, because this process makes it more difficult to maintain and debug the code and decreases its life cycle. Convert only critical functions to assembly code, maintaining a C-like interface.

**7. Testing each converted function and the whole system for the bit-exactness of conversion.** Create output test vectors for each function and for the whole system on a PC- and DSP-simulation environment, using various input test vectors or test cases that cover all possible function states. Comparing these output test vectors bit by bit allows you to test the bit exactness of conversion. You should make sure that output test vectors contain not only the final result/output of the algorithm at each step of simulation but also internal static variables of the algorithm because not all bugs lead

to differences in the final output. Additionally, this step allows you to quickly identify where the assembly code differs from C. Before testing bit exactness of the output vectors that assembly files generate, you must first test whether the C code gives a bit-exact result when you compile the test environment and run it under different compilers and platforms, such as 16- and 32-bit compilers and PC and DSP platforms.

**8. Testing in real time.** You should use the same C code that you used in the PC model (or, for converted functions, their bit-exact DSP assembly version) in the real-life system. The test setup should be as close to real-life conditions as possible, and testing should include as many counterpart devices as possible, for example for interoperability. If you find any problems during real-time testing, development should return to the second step of this process to introduce corrections to C code.

**PROGRAMMING RULES AND PRINCIPLES**

Consider some basic rules and principles for writing DSP software:

- Each header file should have protection against multiple inclusions, as in the following example:  

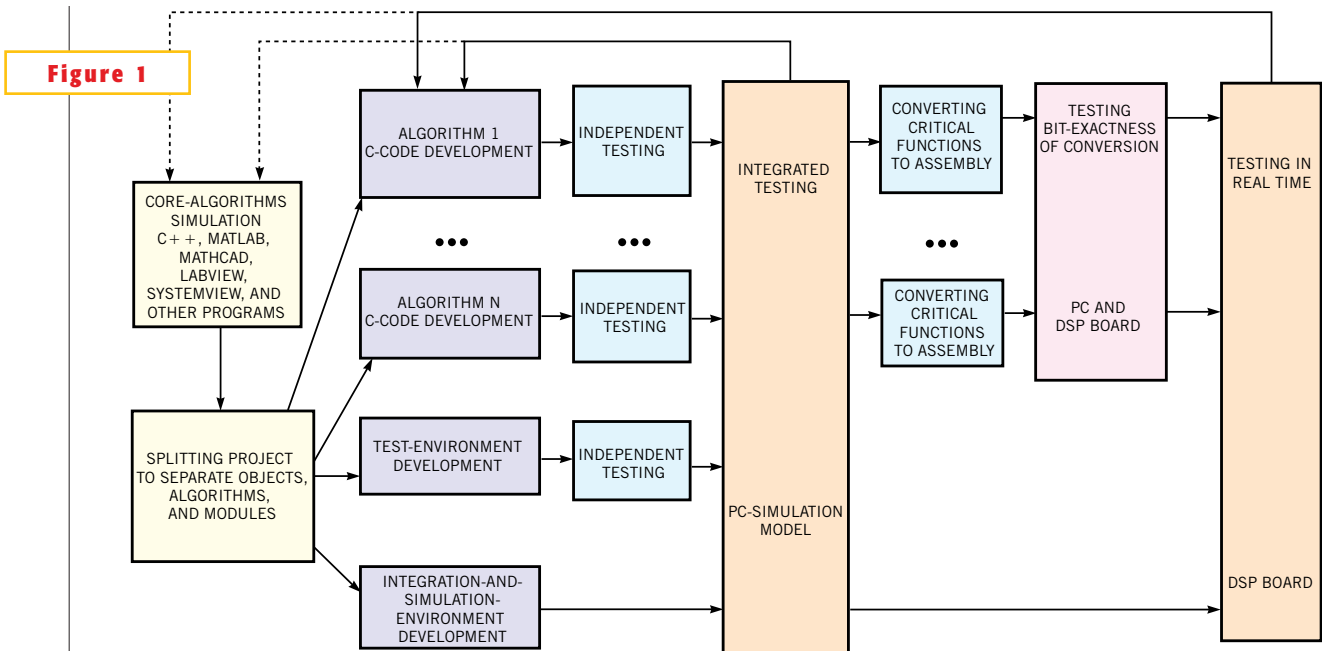
```
#ifndef __STDIO_H
#define __STDIO_H
```

```
...
#endif __STDIO_H
```

- You should declare all global functions and constants in a C file as *extern* in the header file corresponding to this C file. You should hide all functions and constants, local for the given object, using modifier *static*, and in no case should you reference the functions and constants in the header file.
- To use C code in a C++ project, use a substitute of *extern* modifier, for example, macro *EXTERN\_C*, which C will define as just *extern* but C++ will define as *extern "C"*.
- All constants should have, if possible, a *static* modifier and a *const* modifier; otherwise, they will take memory space both in the *.init* section and in the *.data* section. Constants that are local for the given function must have the modifier *static const*. Otherwise, some compilers may create this constant each time you enter this function.
- When defining initialized arrays, you must specify the array's size. In other words, instead of using:

```
const int aFilter[ ] =
{
    1, 2, 100, 2, 1
}
```

use



The DSP-software-development and -testing process comprises many steps, starting with the development and simulation of core algorithms.

```

const int aFilter
[FILTER_SIZE]=
{
    1, 2, 100, 2, 1
}

```

- You should use a *const* modifier to declare references to the objects that a pointer passes to a function and that are remain unchanged inside the function.

## USING OBJECT-ORIENTED PRINCIPLES IN C

The following principles give some hints on how to use an object-oriented approach when writing in C. Although some of the principles may lead to additional overhead, they provide a better code structure.

You can apply these principles both at the level of self-contained algorithms and at the level of simple, small, dependent objects. Self-contained algorithms should conform not only to these principles but also to the xDAIS standard from Texas Instruments ([http://dsp.village.ti.com/docs/algo\\_stand/tms320.algohome.jhtml](http://dsp.village.ti.com/docs/algo_stand/tms320.algohome.jhtml)) so that you can easily integrate these algorithms into other systems. Notice that when you follow these principles, it is much easier to make your algorithms xDAIS-compliant.

Experience shows that object encapsulation, even in a simple form, simplifies code support and maintenance and sometimes even allows saving memory or MIPS. With complex objects, object encapsulation allows the developer to concentrate on the behavioral aspects of the program and not on how each function works.

Note that the following refers to “object definition” as a “class,” even though the C language does not explicitly define this notion.

- A function prototype should contain only those parameters that the function needs. For example, use the modifier *const* in a function-prototype definition for pointers that the code uses only for reading data inside the function; pass a pointer to a smaller object instead of a “global” pointer. Create all modules so that they are as independent and as self-sufficient as possible.
- All functions should be re-entrant. All object variables should reside in an external structure.

- Avoid using global variable and static nonconstant variables.
- Make a pointer to an object, or an instance pointer, the first parameter in the function prototype. You can call this parameter *this* or *This*.
- Each class, or object, must reside in a separate C file, and its header file should contain a description of all its links.
- Developers should try to reuse classes, even the most simple ones.
- Each class must have a constructor.
- Explicitly initialize all variables in the constructor, which should fill a virtual table of methods.
- One object should not explicitly access variables of another object. Each object should have inline functions, or methods, for accessing its variables.
- Even in C, you can have “pseudoinheritance.” The first element of the new class can be the base class; you can redefine a base-class function in a new class as *inline* functions.
- It is desirable to develop objects in such a way that excludes a strict order of how to create/initialize related objects. In other words, you want to be able to create an object in any sequence and initialize the object in any sequence.

## WRITING PORTABLE CODE

You should develop the code to be as platform-independent as possible. The following rules can help you accomplish this goal:

- Use special common types that you can redefine in a certain way on each platform. For example, the following redefinition would be correct on most of the platforms:

```

typedef long int32;
typedef short int16;
typedef unsigned long uint32;
typedef unsigned short uint16;

```
- Declare all variables that need to be 16-bit as *int16*, all 32-bit variables as *int32*, and so on. However, if the size of some variables is unimportant, you should declare them simply as *int* because this declaration usually allows the compiler to better optimize the code.
- Avoid arithmetic operations with type *char*, but rather explicitly convert arithmetic operations to *int*,

because some compilers may perform a signed extension of *char*, and some may perform an unsigned extension.

- In any operation for which the intermediate results may overflow the size of input arguments, you need to explicitly convert the arguments to the type of greater size. Thus, instead of using

```

AInt16=BInt16*CIInt16;
AInt32=BInt16<<5;

```

use

```

AInt16=((int32)BInt16)*((int32)CIInt16);
AInt16=((int32)BInt16)<<5;

```
- Initializing all the variables of the object is highly recommended, for example, using *memset(This,0,sizeof(\*This))*.
- Bitwise shifts should have only positive shift value. Thus, instead of using

```

C=-5;
A=B>>C;

```

use

```

If( C<0 )
    A=B<<-C;
else
    A=B>>C;

```

- Do not declare variables or functions without explicit type definition. Thus, instead of using

```

extern I;
extern GetParam();

```

use

```

extern int I;
extern int GetParam();

```

- You should keep in mind that object, or type, size is measured in the *size\_t* units, not in *int*. Wherever object size needs to be calculated, you should use the *sizeof()* function.

## WRITING C CODE ORIENTED TO DSP

From the early stages, you should write C code oriented to DSP-processor specifics. Creating a set of primitive functions, simulating saturation, circular-buffer operation, and so on is recommended. You can find an example of such a set in *basop.c* file of G.723.1 Recommendation. You should also perform such an operation throughout all the code only through these primitive functions. This step will allow you to keep the logic of the code when porting to assembly.

You should split an object’s variable into at least two structures, for slow and

for fast memory. Variables that the code never uses can simultaneously overlap in the physical memory to save memory space. In C, you can implement this overlap by uniting these variables inside one structure using *union* construction.

You should make delay lines and buffers using circular addressing if applicable. In the case of TI's C54x DSP core, to decrease the loss of memory because of memory alignment that is necessary for circular buffers, you should place buffers in the beginning of "fast" structure, or structure in fast memory, with larger buffers placed first to decrease unused gaps of memory caused by alignment.

#### WRITING FIRM CODE

The term "firm code" implies the ability of the code to resist erroneous actions of the developer and to operate well, even with erroneous input data. The following are the important features of firm code:

- Allow the compiler to output all possible warnings, and make sure

that your project compiles without any warnings, if possible.

- Use *enum* instead of *int* wherever it makes sense.
- Use debug diagnostics. Test all the assumptions that you had in mind while writing the code, even if they are obvious. Testing pointers for equality to NULL is important. For example, you can use the macro *ASSERT()* for this purpose, as follows:
 

```
void SetMemory(int * pBuffer,
int count)
{
    ASSERT(pBuffer!=NULL);
    while(count-- > 0)
        *pBuffer++=0;
}
```
- Wherever possible, use more general test conditions. Thus, instead of
 

```
if(A==256)
    A=0;
```

 use
 

```
if(A>=256)
    A=0;
```

- Make sure to reread what you have written and to step through the code in a debugger.□

---

#### AUTHORS' BIOGRAPHIES

*Nikolay Abkairov, a leading project manager at Spirit Corp (www.spiritcorp.com), is responsible for the development cycle of DSP-software products and manages the company's Fax & Data Modem lab. He has more than four years of experience in this area. He holds an MS in automated control from the Moscow Institute of Electronic Engineering (Moscow) and is currently working on his PhD.*

*Alexey Nazarov, a modem lab project lead at Spirit Corp, is responsible for the development of DSP-software products. He has six years of experience in developing complex software applications for DSP. He holds an MS in radio physics and electronics and a PhD from the Moscow Institute of Power Engineering (Moscow).*