BP30
Bank switching Specification

Edition 2005

Published by Neonseven s.r.l.,
Viale Stazione di Prosecco, 15

34010 Sgonico (Trieste) Italy

Table of Contents

1 Document Mission/Scope

1.1 Mission

This document describes the implementation of bank switching for the BP30 platform, which allows increasing the flash memory size beyond the C166 addressing capability of 16 MB.

1.2 Scope

This document is relevant to all people involved in flash memory extension and configuration for the BP30 platform.

2 List of Acronyms

Abbreviation / Term	Explanation / Definition
BP30	Basic Platform 30
CBE	Common Build Environment
CS	Chip Select
GPIO	General Purpose Input Output
JTAG	Joint Test Action Group
LSB	Least Significant Byte
MB	Mega byte
MSB	Most Significant Byte
OS	Operating System

3 Introduction

The C166 microcontroller allows accessing memory up to 16 MB using a 24 bit address, which is not enough to hold the full-feature SW release of the BP30 platform. In order to overcome such intrinsic limitation an additional flash memory has been added and a bank switch management technique has been implemented in software.
The basic idea is to overlap different code sections in the same address area via locator options and to activate the proper bank before calling their functions. This means that no direct call to these routines is allowed (as banks are not visible to each other): invoking function in banks is only possible via stubs, which perform bank activation, function invocation and bank deactivation.

4 HW design

4.1 HW schematics

The BP30 Globe 6 board is equipped with an additional flash chip of 4 MB, which is under domain of CS4 and is activated by CS4 itself. Such memory is divided into 2 sub-banks of 2 MB each via a GPIO, namely GPIO_31, which is connected to the A21 address pin of the additional flash. The rest of the flash address bus (A1-A20) is directly connected to its C166 counterpart (A0-A19); only A0 is not connected in order to ensure, together with proper BUSCON4 programming (16-bit demultiplexed bus), a memory access by words, i.e. to an even address. With this configuration the SW, in order to access a flash memory location, must first select the sub-bank by setting the proper output value of the GPIO, and then specify its offset from the initial address of the sub-bank. For example the memory location 0x000012 is referenced by setting GPIO_31 output to 0 and by addressing 0x000012, while the memory location 0x20A010 is reached by setting GPIO_31 to 1 and by writing 0x00A010 to the address bus.

Figure 4-1 shows simplified high level HW schematics for the additional flash memory connections to C166 (without considering power supply, reset, clock etc that are not of interest in this context).

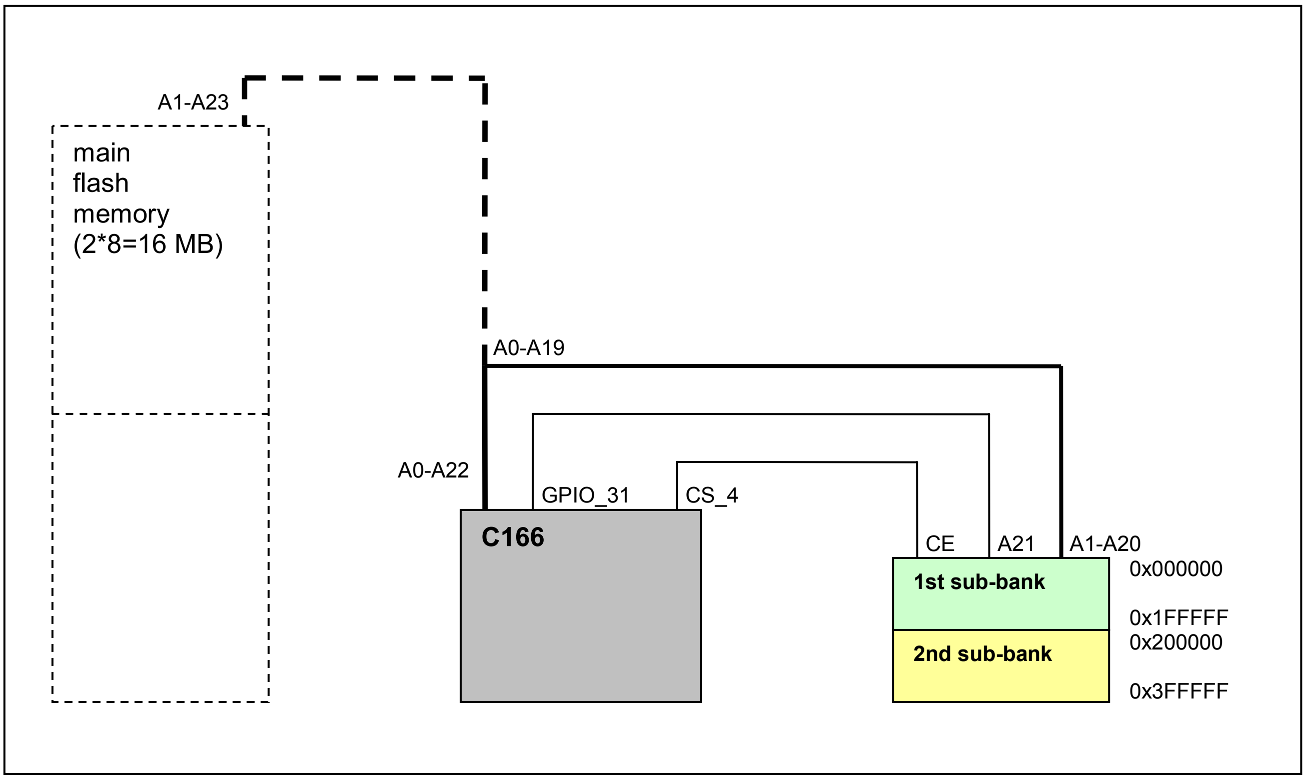

Figure 4-1 HW schematics

4.2 HW limitations

From the HW point of view the most important limitation is that the additional flash memory must be a single chip, separate from the main flash (i.e. it's not possible to have a single chip of 32 MB). Its size may be expanded beyond 4 MB: the main constraint is to keep the allocation scheme of sub-banks of 2 MB each, in order to minimize the shaded area in the main flash memory and the GPIO usage (see next chapter). Each time the number of sub-banks is doubled another GPIO is needed in order to select the proper 2 MB area. The BP30 Globe 6 is designed for extending the additional bank up to 8 MB, with GPIO_31 reserved: further increase is possible provided that the necessary GPIO are available.

5 SW design

5.1 SW functionality

The C166 microcontroller can only access up to 16 MB of memory; an increase beyond this limit requires the implementation of memory banking, which consists in mapping different code sections belonging to different flash banks onto the same logical address area (via locator controls) and then activating the corresponding physical bank prior to access the hosted data and/or code. This implies that the code located in banks is not directly accessible any more because bank activation must be done beforehand by the so-called function stubs, which on the contrary must be always visible and therefore must be located outside the common overlaid address area of 2 MB.
This scheme leads to the logical memory map represented in Figure 5-1: the only difference with previous configurations is the new overlay area between 12 and 14 MB, now reserved for memory banks and previously allocated for visible code and data.

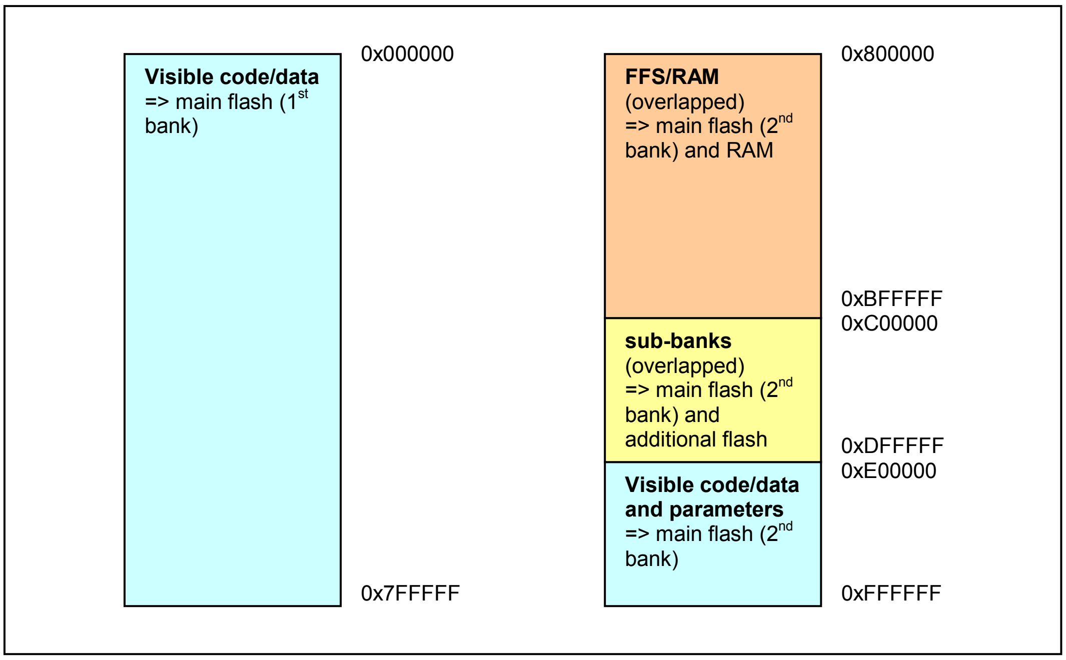
Visible code/data => main flash (1st bank)
0x000000
0x7FFFFF
FFS/RAM (overlapped) => main flash (2nd bank) and RAM
0x800000
0xBFFFFF
0xC00000
sub-banks (overlapped) => main flash (2nd bank) and additional flash
0xDFFFFF
0xE00000
Visible code/data and parameters => main flash (2nd bank)
0xFFFFFF

Figure 5-1 Memory map

The choice of location and size for the overlay area comes from the following considerations:
<u>Location</u>: bank switching slows down the execution speed; therefore it's worth placing banks in an area (the second flash memory) that is already slower as page mode is not used, thus leaving the faster first chip for critical code. AS the 8-12 MB range is already reserved for FFS and RAM and the last sectors for parameters, only the 12-14 MB area is available.
<u>Size</u>: increasing the bank size reduces the number of GPIO necessary for bank selection but increases the shaded area in the main flash and therefore reduces the memory available for data and visible code. 2 MB is a reasonable trade-off between these 2 opposite requirements.

There are plenty of techniques for implementing bank switching: the choice was made according to both technical and practical considerations (see BH02.S1.TN.000005). The main ideas are:

1. Minimizing the source code modifications and doing most of the job by <u>post-processing</u> intermediate <u>.src</u> files: in this way it's easier and faster to integrate either new SW versions or new SW modules, including third-party libraries, without involving any developer (apart from the system integrator).
2. Locating only <u>pure code in banks</u> and leaving all data (including bitmaps, resources etc) in the visible area: this avoids modifications to the (scattered) code that actually performs data access, which may change in the future and would require for each new version a specific and separate branch in Clear Case with the necessary C166 adaptations.
3. Providing <u>configuration options</u> to the system releaser, as there may be different SW feature sets for different products based on the same BP30 platform; therefore it must be always possible to select which libraries to locate in banks on a project basis.

5.2 SW implementation

Bank switching implementation involves the following three main areas: function stubs, operating system and Apoxi kernel.

5.2.1 Function stubs

For all functions located in banks a corresponding function stub must take responsibility for activating the proper bank, calling the desired function then restoring the previous bank. As a consequence all calls to routine located in banks are replaced by calls to their stubs, which of course must always be accessible in order to allow inter-

bank function calls and for this reason they're located in the visible memory area. Stubs generation is performed via a Perl script that performs post-processing of the intermediate assembly file generated by the C166 Tasking compiler. Anyway only functions longer than their stubs are worth processing; otherwise we would have no advantage from placing them in banks.
Stubs implementation is based on two main concepts: the segment trick and the callee strategy.

5.2.1.1 Segment trick

In order to locate different code sections onto the same overlapped memory area the OVERLAY locator control may be used; however this would make debugging difficult because not only source code references but also assembly labels would be lost. A better solution that preserves debug information is the so-called "segment trick": it consists in locating the different sections in different memory areas having in common the LSB of the segment number (exploiting the MEMORY and CLASSES locator controls) then using the segment MSB for representing bank information, used to activate the bank, and the segment LSB as the segment number, used to perform an indirect function call. In this way

1. From the logical point of view the different sub-banks are located in the memory areas 12-14 MB, 28-30 MB, 44-46 MB (i.e. 12+16*bank_id to 14+16*bank_id, bank_id=0,1,2,…).
2. From the physical point of view (.hex file) the sub-banks are stacked from 16 MB on in 16-18 MB, 18-20 MB (16+2*(bank_id-1) to 16+2*bank_id, bank_id=1,2,…): this is achieved by post-processing the .hex file and moving the sub-banks.
3. From the execution point of view all sub-banks are overlapped in the common overlay area between 12 and 14 MB as function calls are performed using the LSB of the segment number.

Example: a function located at address 0x1C0A3F8 is called by activating bank 1 (segment MSB = 0x1000000) then jumping to the address 0x0C0A3F8 (segment LSB = 0xC0 + offset = 0xA3F8) within the overlay area.

Figure 5-2, 5-3 and 5-4 represents the three different points of view listed above.

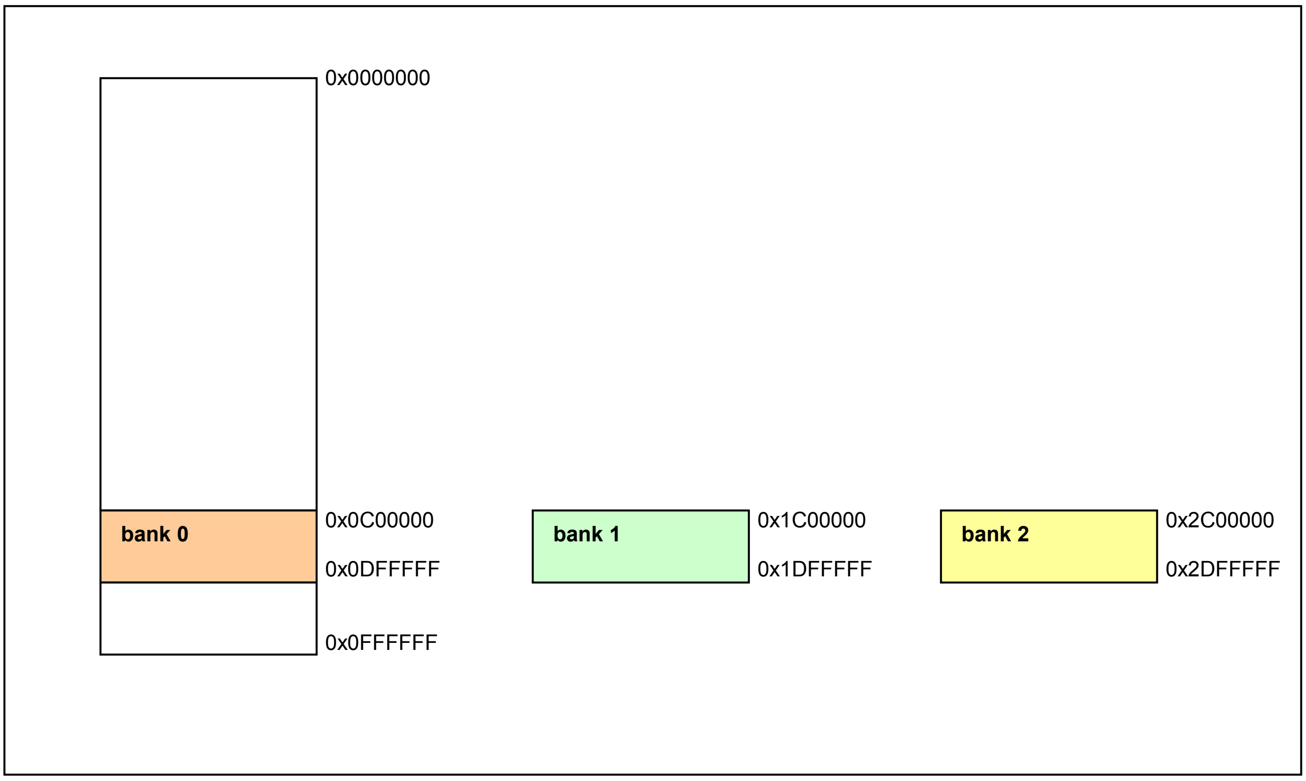

Figure 5-2 Logical location.

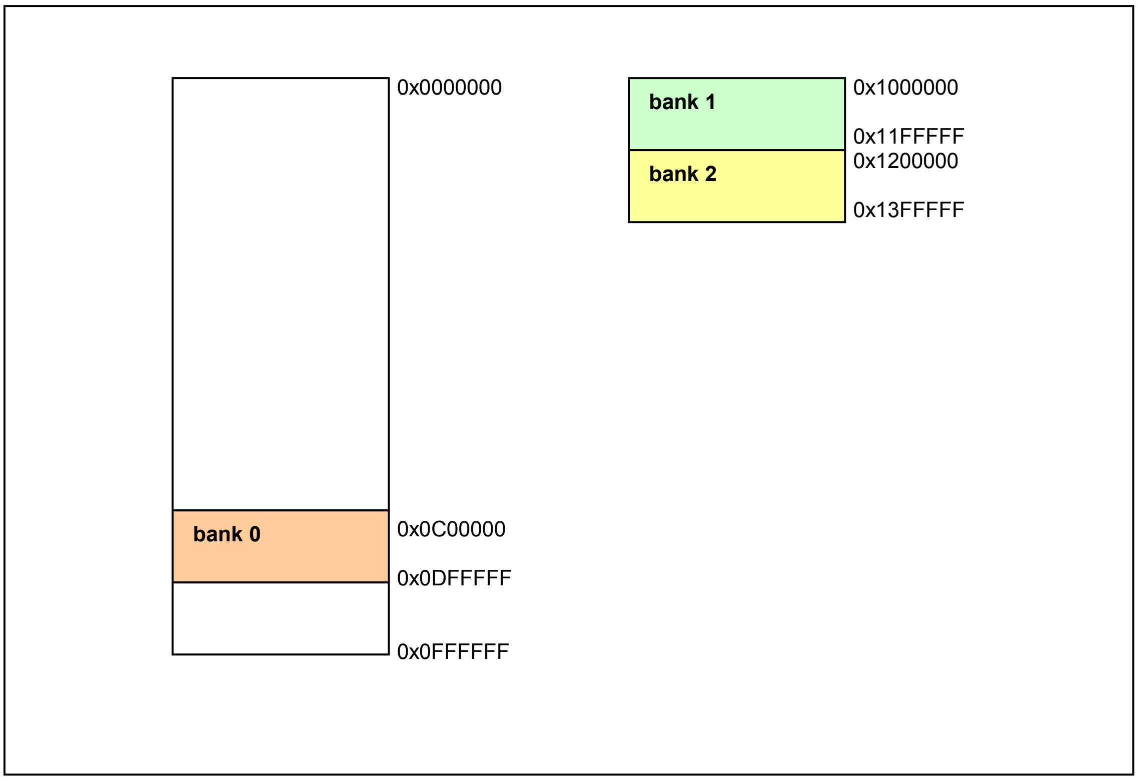

Figure 5-3 Physical location.

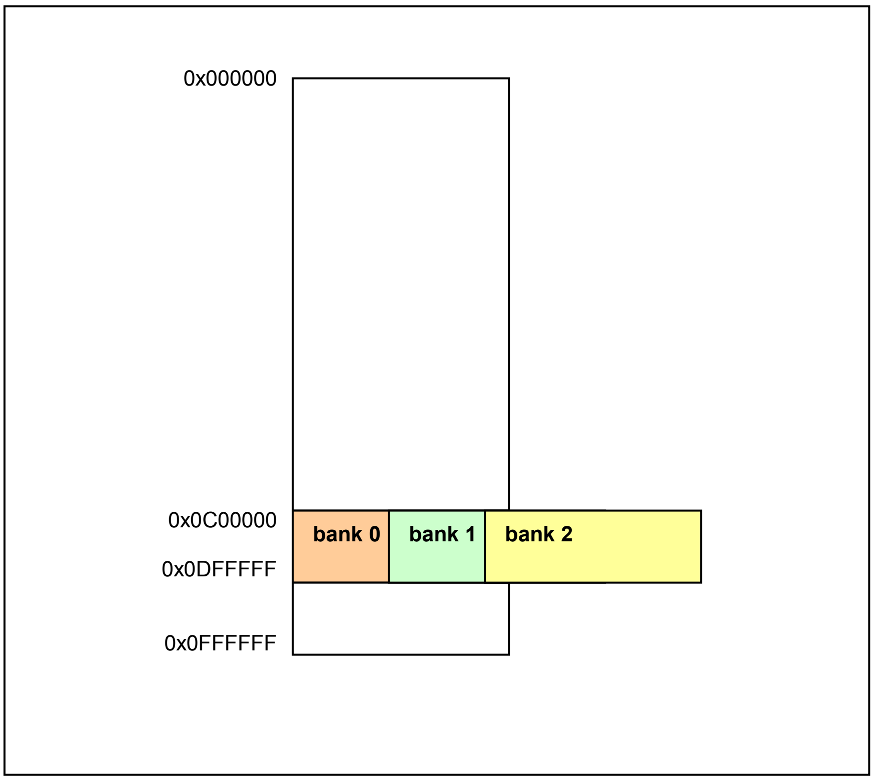

Figure 5-4 Execution location.

Figure 5-5 shows the listing of a typical function stub with the usage of the segment trick.

```
          PUBLIC  _GetInstance__8Bank1AppSFv
_GetInstance__8Bank1AppSFv          PROC     FAR

          MOV R1,#SEG __GetInstance__8Bank1AppSFv          #Load the segment number
          SHR R1,#08h                                      #Retrieve the MSB, representing the bank ID
          CALLS SEG ACTIVATE_BANK,ACTIVATE_BANK            #Activate the bank whose ID is R1
          PUSH R1                                          #Save the old bank ID (returned in R1)
          MOV R4, #SOF __GetInstance__8Bank1AppSFv         #Load the function address offset
          MOV R5, #SEG __GetInstance__8Bank1AppSFv         #Load the function address segment
          AND R5, #00FFh                                   #Mask the segment MSB
          CALLS SEG __icall,__icall                        #Perform an indirect call to the overlay area
          POP R1                                           #Retrieve the original bank ID
          CALLS SEG ACTIVATE_BANK,ACTIVATE_BANK            #Restore the previous bank
          RETS
```

Figure 5-5 Function stub code

5.2.1.2 Callee strategy

Replacing all functions located in banks with their stubs is very difficult to do, especially taking into account virtual functions and indirect calls. It's also computationally heavy because it requires building a huge database of all the references to functions located in banks, then performing a global search and replace all over the .src files and finally executing a further compile and link stage.

A better solution is the so-called callee strategy: it consists in leaving all calls to function in banks as they are but replacing their body with their stubs. Their body is moved into another function, having a different name (built from the original one by adding a leading underscore) and called by the stub only. This algorithm provides efficiency and flexibility because only functions located in banks are over-headed with a stub while the rest of the code is not affected at all and does not even need to be re-compiled when enabling/disabling memory banking. Moreover symbolic debug information is entirely preserved. The only drawback is the increased code size as each function located in banks has its own stub, while replacing the caller would have allowed having a single common stub function for bank activation.

5.2.2 Operating System

OS task switches may interfere with banking not only because of different priorities (currently all Apoxi containers have the same priority level) but also due to the usage of semaphores and mutexes; for this reason OSE must be adapted accordingly. This requires adding a new variable to the process data structure and updating it during process management. Unfortunately conf166.exe ENEA tool does not use the definitions in os166.asm when generating the os166.tmp file containing the process control blocks, but creates a fixed-width data structure; therefore a post-processing of os166.tmp is needed to make room for the process bank id. To sum up, the OS is modified as follows:

1. os166.asm: whenever an OS context switch occurs, the current bank configuration (i.e. the active bank id) is now saved and later restored before returning control to the suspended process.
2. os166.tmp: is now post-processed in order to add the bank id field to the data structure of each process.

5.2.3 Apoxi kernel

Apoxi kernel is largely based on OSE semaphores and the possible consequent task switching; therefore in most cases (like Application scheduling, messaging, Apoxi semaphores and derivatives, timers) it's not directly concerned by bank switch management. Anyway there is a critical scenario, related to the Active Wait mechanism, which may trigger an Application switch without involving OSE. Given two Applications in the same Container, let's suppose that

1. Application 1 performs an Active Wait after calling a function located in BANK1;
2. then Application 2 performs an Active Wait after calling a function located in BANK2;

3. finally Application 1 returns from the Active Wait with BANK2 still active even though it expects BANK1 active.

In this scenario without Apoxi kernel modifications Application 1 would be resumed with a wrong bank active!
Of course it would be possible to modify the Active Wait mechanism, making it aware of banking to cope with this use case. The impact would be minimized but possible new similar features may exhibit the same behaviour and would require the same modifications again. Moreover we should distinguish between EGOLDlite and SGOLDlite platforms (as only the former requires banking), mixing high-level code with HW related nuances.
For these reasons the setjmp/longjmp function pair has been adapted instead: the overhead is bigger and banking is handled even when it's not necessary, but modifications are encapsulated in code that is already platform-specific and any new feature based on the current implementation of Apoxi Scheduler is already protected from banking issues.

5.3 SW limitations

The main SW limitations of this specific memory banking implementation are:

1. Only 10 MB of flash memory is available for data, parameters and visible code (the first 8 MB and the last 2 MB); the 12-14 MB address range is reserved for banks;
2. Only pure code is located in banks: data must always be visible and therefore located outside banks;
3. In order to locate a library in banks the intermediate .src files must be available for post-processing: this rules out the possibility to locate Java in banks for example, although it would be an ideal candidate, and in general any third-party library provided in object code only;
4. Debugging the code in banks with Lauterbach JTAG is difficult because Trace32 is not aware of banking, therefore source level code information is inevitably wrong.
5. Code may be located in banks on a library basis only: handling single files would be too complex for the CBE.
6. Each function located in banks must have its own stub: this increases the global code size and also the stack size, possibly provoking stack overflows if stack allocation is too strict and does not have proper margins.

6 References

6.1 External

[1] OSE for C166 Kernel User’s Guide, OSE Systems Inc, 2001.
[2] OSE for C166 Kernel Reference Manual, OSE Systems Inc, 2001.
[3] C166/ST10 v7.5 cross-assembler, linker/locator, utilities user's guide, Tasking Inc, 2001.
[4] C166/ST10 v7.5 C cross-compiler user's guide, Tasking Inc, 2001.

6.2 Internal

Title	Doc ID
Memory extension for BP2-X	BH02.S1.TN.000004
BP2-X memory banking options	BH02.S1.TN.000005

7 Document change report

	Change Reference		Record of changes made to previous released version	
Rev	**Date**	**CR**	**Section**	**Comment**
1.0	01/08/2005	N.A.		Document created

8 Approval

Revision	Approver(s)	Date	Source/signature
1.0	Stefano Godeas	02/08/2005	Document stored on server