# Bank switching presentation

BANK SWITCHING PRESENTATION

## Memory banking purpose

- Overcome the limited addressing capability of C166 (16MB) in order to cope with the increased code size of the full-feature BP30 SW release.
- BP30 Globe 6 board is equipped with an additional 4 MB flash chip.
- Basic idea: overlap different code sections in the same address area via locator options and activate the proper bank before calling any function in these sections.

## Additional flash

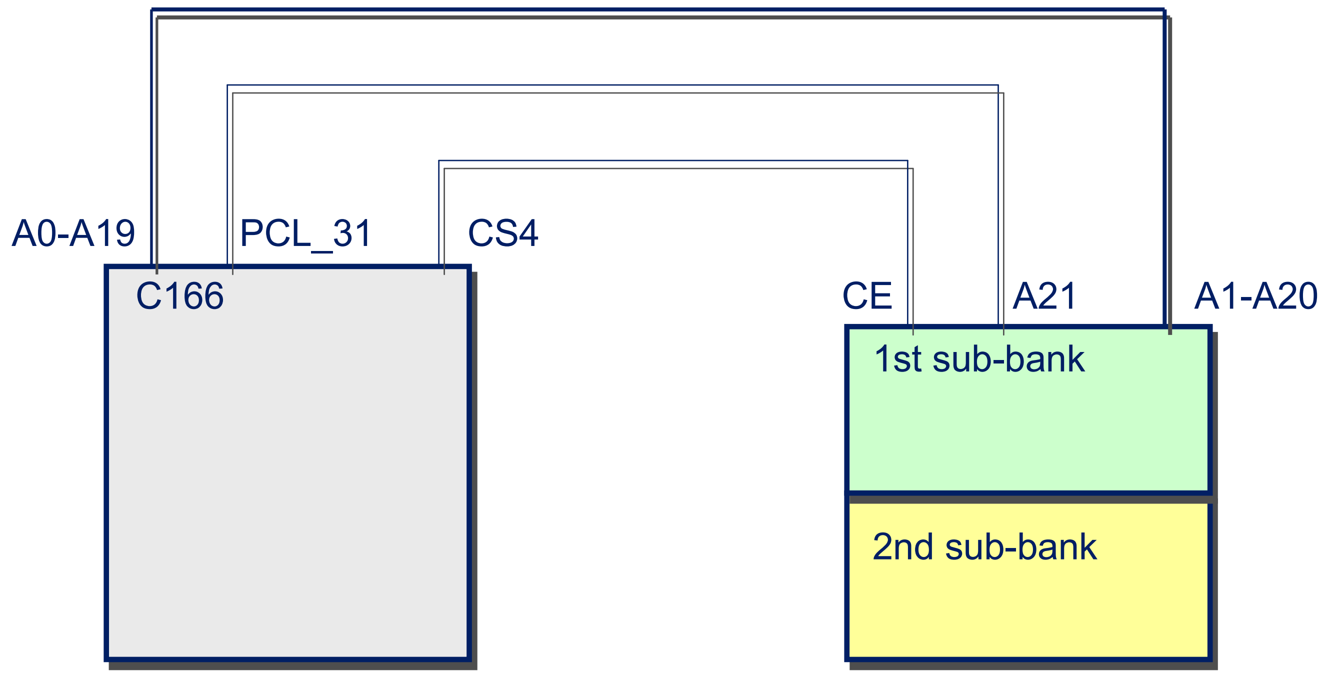
- Under domain of CS4 and activated by CS4.
- Divided into 2 sub-banks of 2 MB selected via GPIO 31, connected to the flash A21 address pin (A1-A20 directly connected to C166 A0-A19).

Additional flash memory access requires:

1. selecting the sub-bank GPIO 31 output value
2. specifying its offset from the sub-bank initial address.

Examples: 0x000012 is accessed with PCL_31 = 0 and by addressing 0x000012, 0x20A010 by PCL_31 = 1 and by addressing 0x00A010.

## HW schematics



## HW limitations

- 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, provided that the allocation scheme of 2 MB sub-banks is preserved.
- Each time the number of sub-banks is doubled another GPIO is needed in order to select the proper 2 MB area (BP30 Globe 6 is designed for memory extension up to 8 MB).

## Memory banking

- Consists in mapping different code sections belonging to different flash banks onto the same logical address area and then activating the corresponding physical bank prior to access the hosted data and/or code.

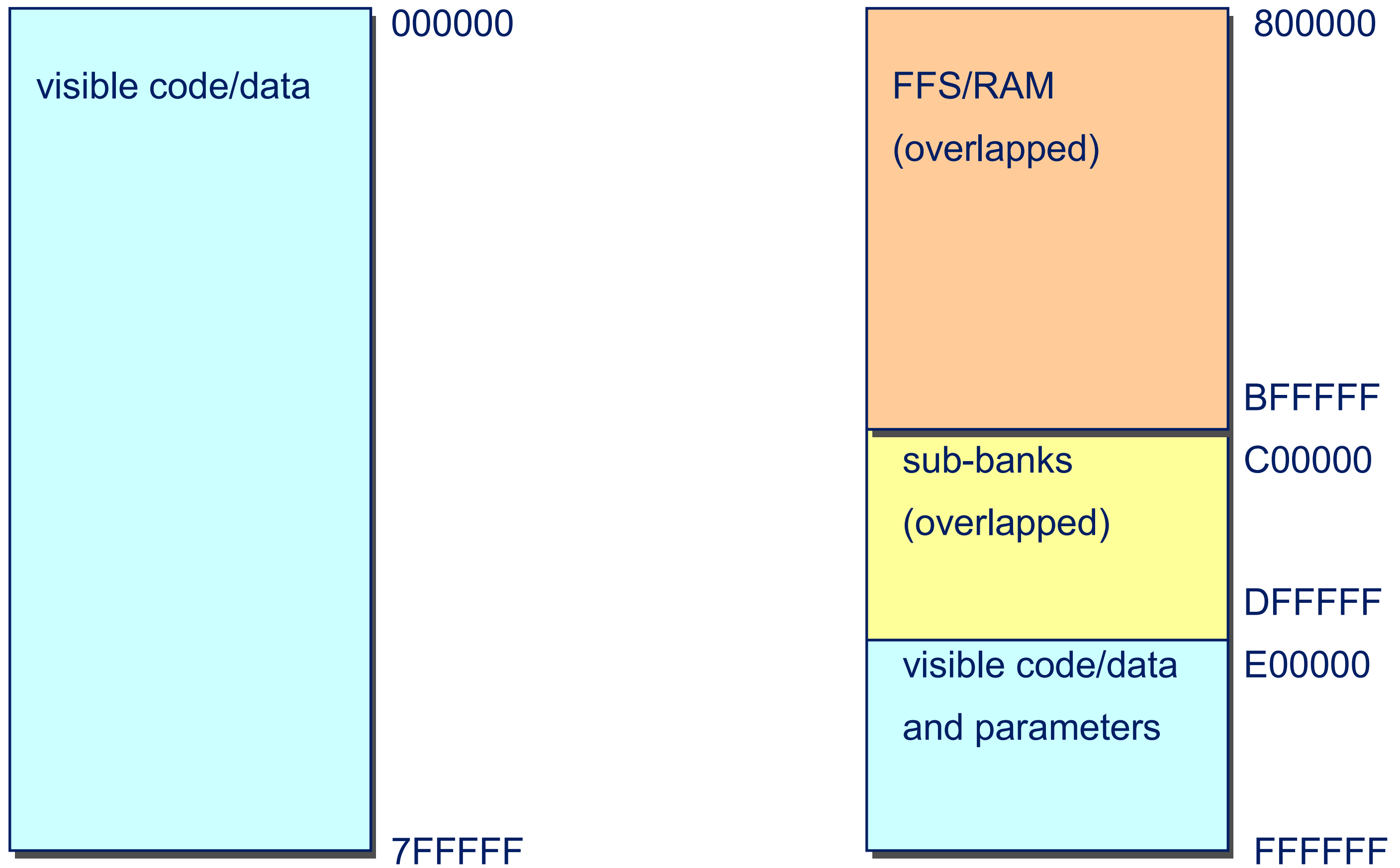
- Code located in banks is not directly accessible any more because bank activation must be done beforehand by the so-called function stubs, always visible and located outside the common overlaid address area of 2 MB.

## Bank location and size

- Location: bank switching slows down the execution speed; therefore banks are located in the second flash memory (no page mode). 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 for banking.

- Size: 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.

## Memory map


visible code/data
000000
7FFFFF
FFS/RAM
(overlapped)
800000
BFFFFF
sub-banks
(overlapped)
C00000
DFFFFF
visible code/data
and parameters
E00000
FFFFFF

## Implementation guidelines

- Minimizing the source code modifications and doing most of the job by post-processing intermediate .src files (easier and faster integration).

- Locating only pure code in banks and leaving all data in the visible area, thus avoiding modifications to the (scattered) code that actually performs data access.

- Providing configuration options to the system releaser, as there may be different SW feature sets for different products based on the same BP30 platform.

## Function stubs

- For any 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.

- All calls to routine located in banks are replaced by calls to their stubs, which must always be accessible in order to allow inter-bank function calls and therefore located in the visible memory area.

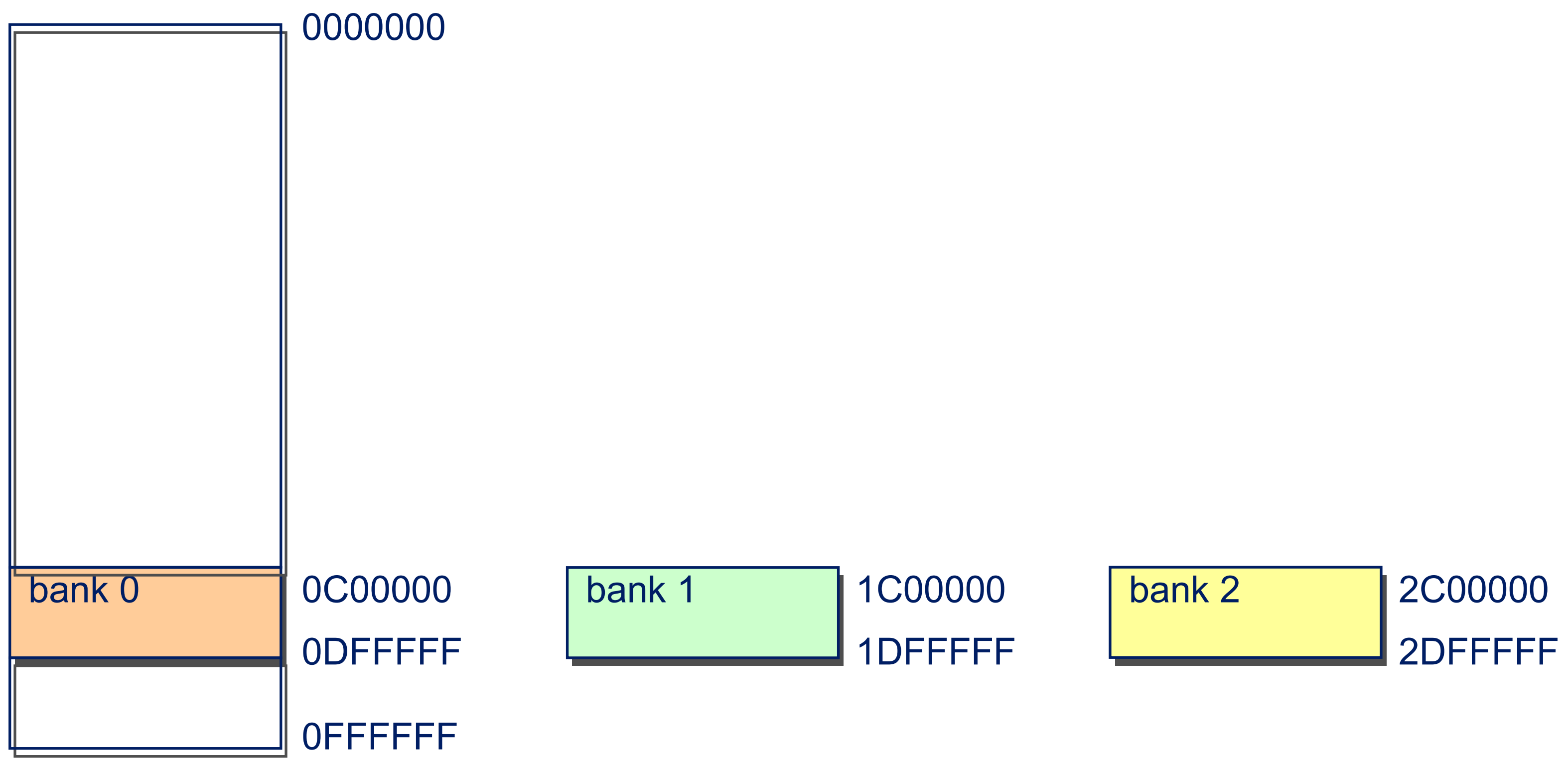
- Stubs generation is performed via a Perl script that performs post-processing of the intermediate assembly file.

## Segment trick

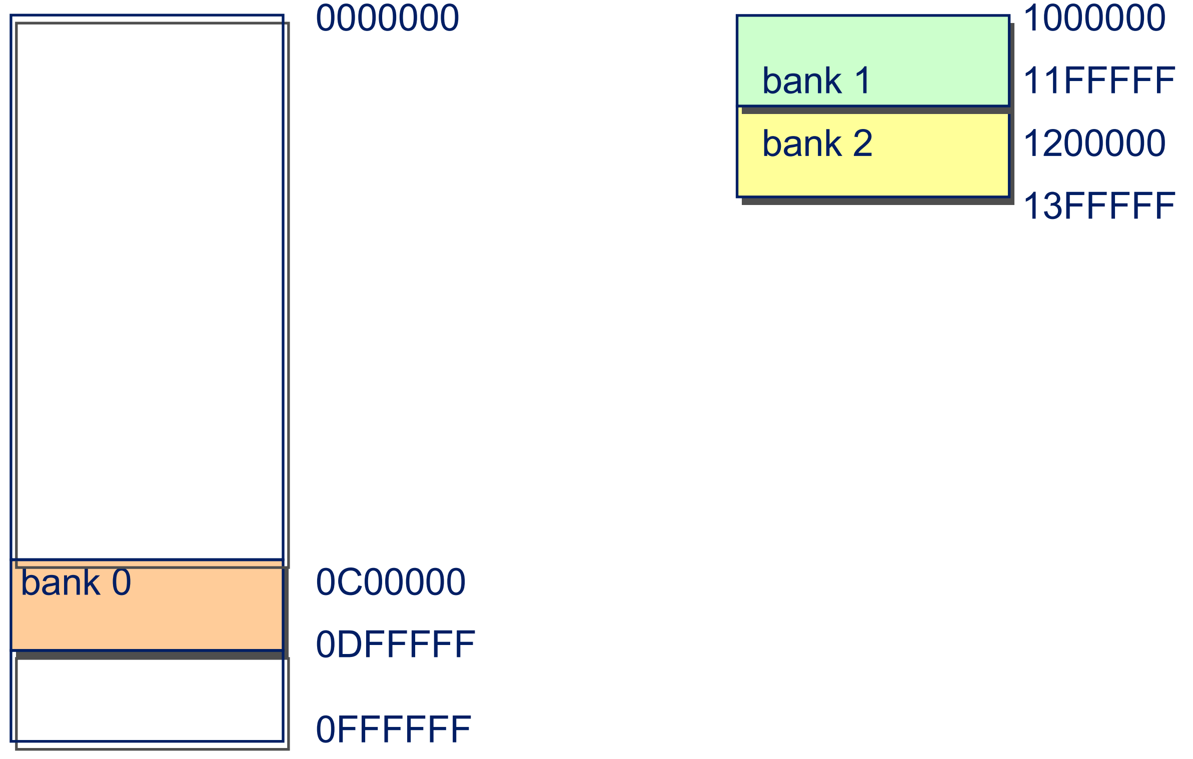
- Consists in locating the different sections in different memory areas having the same segment number LSB then using the MSB as bank id (to activate the bank) and the LSB as the segment number (to perform an indirect function call).

1. From the logical point of view the different sub-banks are located in the memory areas 14-16 MB, 28-30 MB (i.e. 16+12 to 16+14), 44-46 MB (i.e. 32+12 to 32+14 MB).
2. From the physical point of view (.hex file) the sub-banks are stacked from 16 MB on (i.e. 16-18 MB, 18-20 MB): 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.

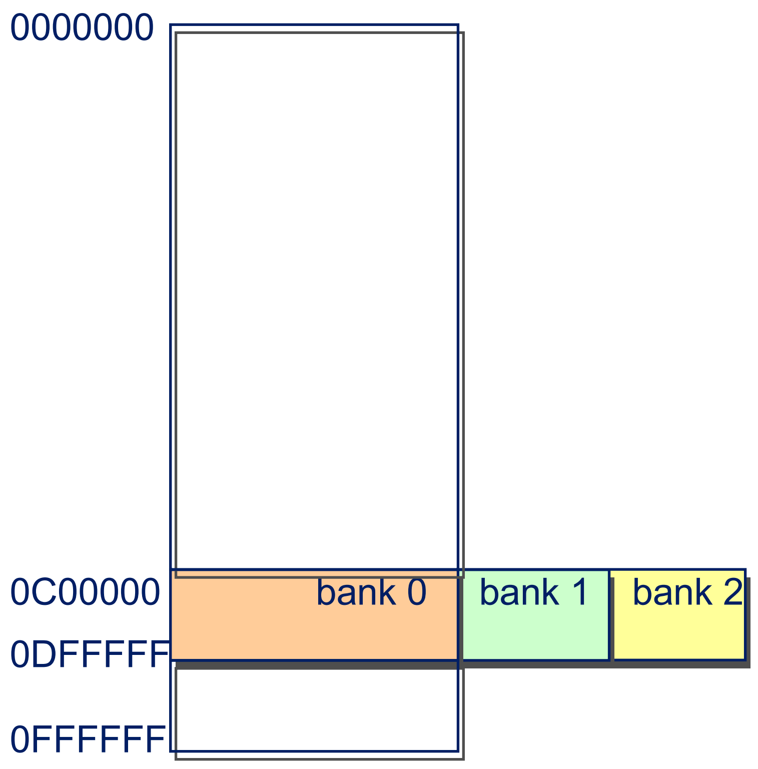
## Logical location



## Physical location



## Execution



## Callee strategy

- Replacing functions with stubs is

1. difficult due to virtual functions and indirect calls and
2. computationally heavy as it requires a huge database of all banked functions references, a global search and replace all over the .src files and a further compile and link stage.

- Callee strategy: consists in replacing the banked function bodies with their stubs and moving the body into another function whose name is built from the original one by adding a leading underscore.

## Callee strategy advantages

1. efficient and flexible because only functions located in banks are over-headed with a stub (the rest of the code is not affected)
2. No need for re-compiling the code outside banks when enabling/disabling memory banking.
3. Symbolic debug information is entirely preserved.

- The drawbacks are the increase of both code size and system stack size, as each function located in banks has its own stub.

## Operating system

- OS task switches may interfere with banking because of different priorities, semaphores and mutexes and must be adapted. OS modifications:

1. os166.asm: whenever an OS context switch occurs, the current bank configuration (i.e. the active bank id) is saved and later restored before returning control to the suspended process.
2. os166.tmp: is post-processed in order to add the bank id field to the data structure of each process (conf166 generates a fixed process control block).

## Apoxi kernel – critical scenario

- Apoxi kernel is largely based on OSE semaphores; therefore in most cases (Application scheduling, messaging, Apoxi semaphores and derivatives, timers) it's not directly concerned by bank switching. 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:

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.

- Without Apoxi kernel modifications Application 1 would be resumed with a wrong bank active!

## Apoxi kernel - modifications

- Modify the Active Wait mechanism, making it aware of banking : minimized impact but risky for possible new similar features with the same problem and specific branch for EGOLDlite family.

- Modify setjmp/longjmp functions: bigger overhead but modifications encapsulated in platform-dependent code and ready to support banking for any new feature based on Apoxi Scheduler.

## SW limitations

- 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;
- Only pure code is located in banks: data must always be visible and therefore located outside banks;
- intermediate .src files must be available for post-processing: this does not happen for Java and most third-party libraries;
- Debugging the code in banks with Lauterbach JTAG is difficult because source level code information is lost.
- Code may be located in banks on a library basis only: handling single files would be too complex for the CBE.
- Function stubs increase the global code size and stack size, possibly provoking stack overflows if stack allocation is too strict.

## References

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.
5. Memory extension for BP2-X, NeonSeven SrL, 2005.
6. BP2-X memory banking options, NeonSeven SrL, 2005.
7. Bank switching Specification, NeonSeven SrL, 2005.