SGS-THOMSON MICROELECTRONICS

APPLICATION NOTE

BANKSWITCH AND GNU C EXAMPLE

by Thierry Crespo and Marc Liochon

INTRODUCTION

Some ST9 devices offer the possibility of accessing up to 2x8 Mbytes of external memory by means of a technique known as ”Bank Switching”, which uses 32 Kbyte memory blocks selected through port 2. This port register defines the bank number for external memory. All 8 bits of this register may be used for bank switching, alternatively the low nibble may be used to address memory and the high nibble may be used as application I/O.

This application note provides an example of how the bank switching mechanism may be handled, and of how it may be run on an ST905x emulator, using the GNU C Toolchain and the WGDB9 Windows GNU Debugger.

In the first part, it is assumed that all 8 bits of the bankswitch register are used for bank switching (normal mode).

An example of bank switching in nibble mode is also provided.

TABLE OF CONTENTS

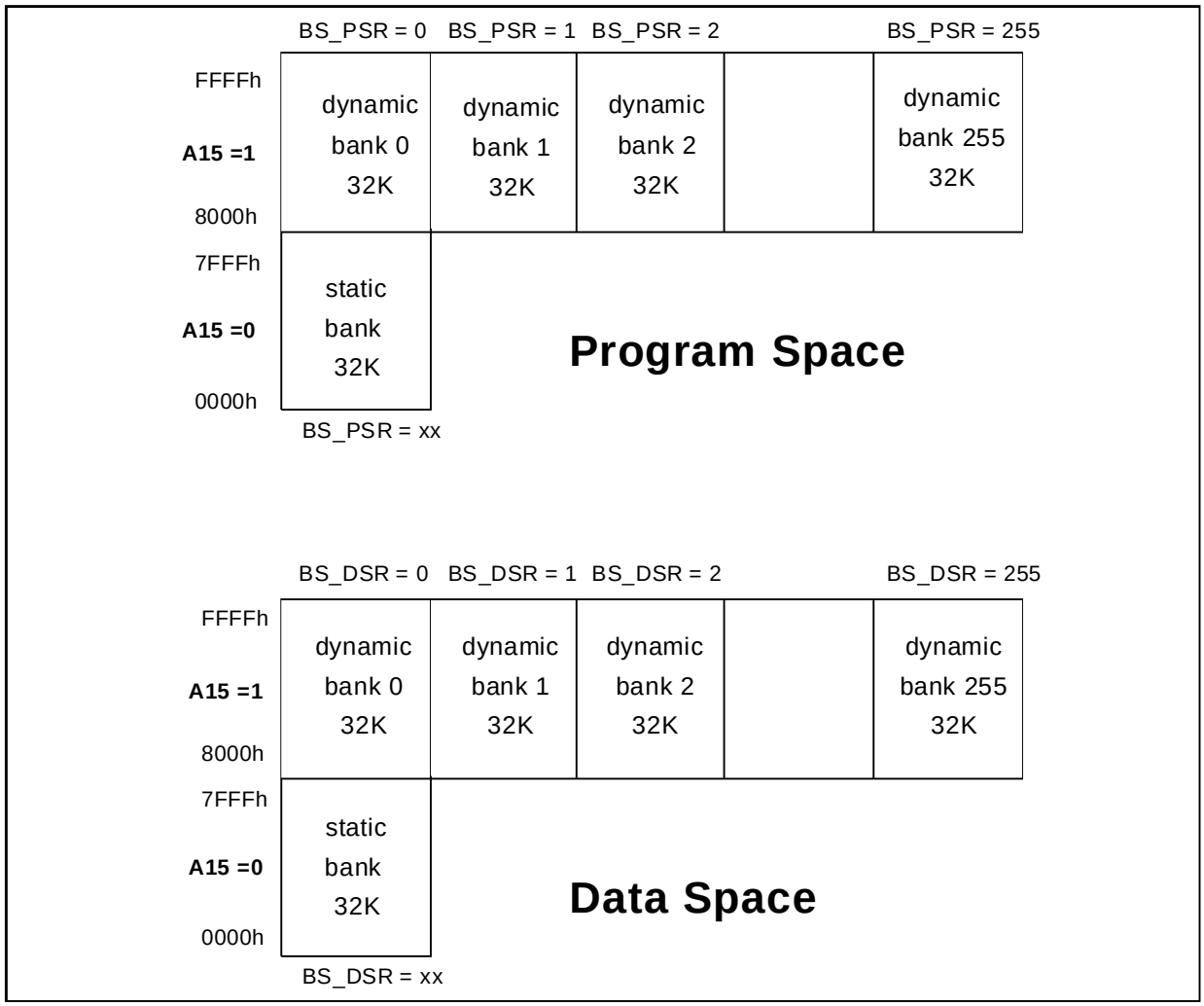
1 THE BANKSWITCH MECHANISM

The ST9, being a 16-bit microcontroler, is normally limited to 16-bit data and addresses. A special feature has thus been implemented in order to be able to access up to 2x8 Mbytes of extended memory (which is equivalent to a 24-bit address), while still using 16-bit addresses. Memory is mapped as 32Kbyte banks:

– 1x32Kbyte static bank for program space
– 1x32Kbyte static bank for data space
– 256x32Kbyte dynamic banks for program space
– 256x32Kbyte memory banks for data space.

This 24-bit addressing scheme is implemented by outputting the 8 most significant bits of the address on the port 2 register, and the 16 least significant bits on the normal address ports.

Figure 1. Bankswitch mapping



When bankswitching is used:

– P2DR = R226 (port 2 data register) contains the 8 most significant adress bits for data space; this register is also named BS_DSR (BankSwitch Data Space Register).

– R227 (port 2 program register) contains the 8 most significant adress bits for program space, this register is also named BS_PSR (BankSwitch Program Space Register).

Then if data space is selected, BS_DSR will be output as A23-A16, and, if program space is selected, BS_PSR will be output as A23-A16.

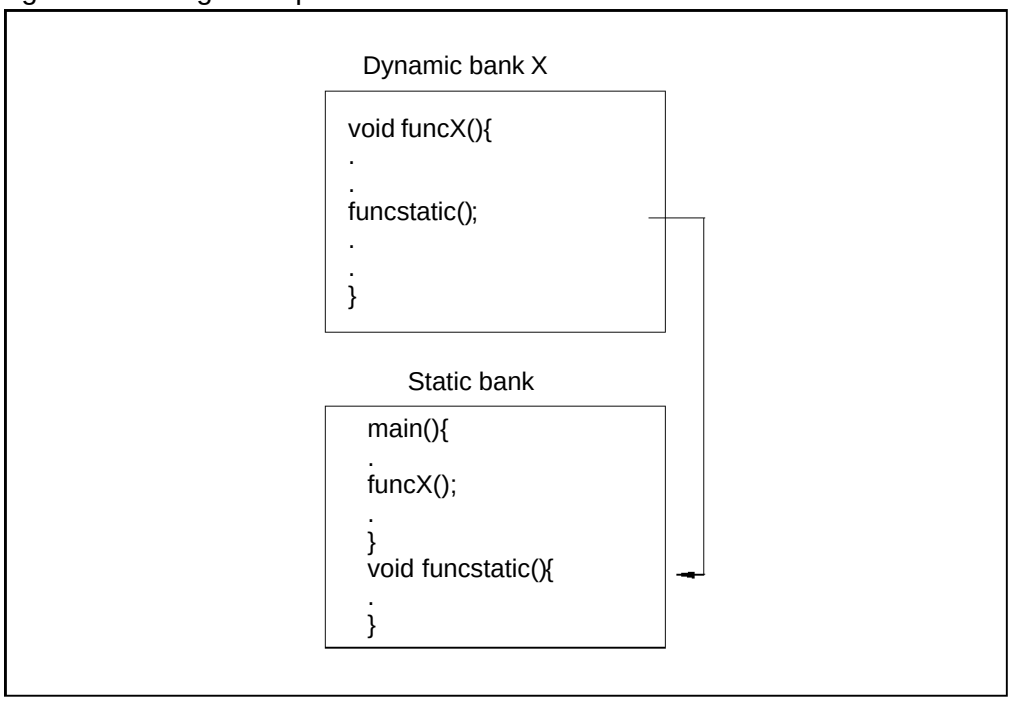
	A23--A16	**A15--A8**	**A7--A0**
Data Space	Port 2 = R226	Port 1	Port0
Program Space	Port 2 = R227	Port 1	Port0

1.1 Accessing the static bank

The static bank, from 0000h to 7FFFh, is determined by A15 = 0. When A15 = 0 the high address (A23-A16) is not output. Thus, whatever the bankswitch register value, the access will be between addresses 7FFFh and 0000h in the static bank.

It is then obvious that calls to this static bank will always be achieved without having to change the high order address bits.

Taking the following example:



A call from code located in dynamic bank 2 (or any dynamic bank) to code located in the static bank, can then be achieved without paying any particular attention to the bankswitch registers.

Care should be taken, however not to change the bankswitch register (R227 = BS_PSR) before returning to the dynamic bank, since in this case the return address could be to the wrong dynamic bank.

1.2 Accessing the dynamic banks

Access to a dynamic bank is determined by A15 = 1. When A15 = 1, the high address (A23-A16) is output from the port 2 bankswitch register and the bank is selected according to the contents of this register.

Lets consider the following example:

	A23-A16	A15-A0	
A15 = 0	05h	70AAh	Static bank
A15 = 1	05h	F0AAh	Dynamic bank 5

This example shows that, if A15 goes from low to high, the static bank is no longer accessed, and a dynamic bank, defined by the value of the port 2 register (A23-A16) is then addressed.

Thus, if code in the static bank makes a call to code in a dynamic bank, the bankswitch register for the dynamic bank needs to be set prior to the jump.

If code in a dynamic bank makes a call to code in another dynamic bank, changing the bankswitch register will automatically change the addressed bank and the call will not be possible. Thus, the only way to make the call is to save the current bankswitch register value in the static bank (which is always accessible from any other bank), jump to the static bank, set the new bank, and then make the call to the code in the other dynamic bank. A return from the call must also be implemented in the same way.

1.3 Mapping code in dynamic banks

In order to achieve gains in speed and code size, access to the dynamic banks will be coded in 16 bits only, allowing faster access and shorter opcodes. The only price to be paid for this gain is that the user must change the high order address (A23-A16) in software (by writing to the port 2 register).

To implement efficient code, the programmer must consider that by minimising the number of jumps or calls between dynamic banks he will maximise code efficiency. An efficient way to manage this is by organising the code in a dynamic bank in such a way that calls are keep internal to the bank as far as possible. Functions which are most frequently used should be mapped in the static bank, so that no change is required on the high order address bits.

2 USING THE BANKSWITCH WITH GNU C

This chapter describes how to use the bankswitch feature with GNU C.

In the following description, we will refer to a call performed from one bank to another as a ”far call”.

2.1 Bankswitch management in C

In C, the previously described operations are totally transparent to the user, since the compiler manages far calls automatically. The compiler generates the code required for handling the bankswitch registers.

It is, however, necessary to issue directives to the compiler, to instruct it that far calls will be processed.

A very short example will now be shown to explain how functions mapped in different banks should be declared and used, and what the compiler does.

The program sequence is as follows:

1) enter the main procedure (static bank)

2) inside main, call func0 (call from the static bank to a dynamic bank)

3) inside func0, call func1 (call from a dynamic bank to another dynamic bank)

Given the following files:

bank0.c

```
#pragma far (func0)
extern void func1();

void func0(){
  func1();
}
```

bank1.c

```
#pragma far (func1)

void func1() {
  asm(”nop”);
}
```

main.c

```
extern void func0();

void main(){
  func0();
}
```

Where:

– bank0.c is a portion of C code mapped in dynamic bank 0,

– bank1.c is a portion of C code mapped in dynamic bank 1,

– main.c is is a portion of C code mapped in the static bank.

”**#pragma far**” is the directive which instructs the compiler that a function will be mapped in a dynamic bank.

When the compiler finds such a directive, it generates a small function called a **stub** function, always mapped in the static bank, which will be used to manage the far calls. This stub function is stored in a file called **_static_.s**, automatically generated by the compiler, but which needs to be compiled and linked with the application by the user.

The _static_.s file contains the macros to manage calls.

Compilation and linking are described below.

The following assembly files will be generated:

bank0.s9

```
f$func0:
 .far
 pushw rr12
 ldw rr12,RR238
 call func1
L1:
 ldw RR238,rr12
 popw rr12
 jx __ret_far
```

bank1.s9

```
f$func1:
 .far
 pushw rr12
 ldw rr12,RR238
 nop
L1:
 ldw RR238,rr12
 popw rr12
 jx __ret_far
```

main.s9

```
main:
 pushw rr12
 ldw rr12,RR238
 call func0
L1:
 ldw RR238,rr12
 popw rr12
 ret
```

static.s9

```
__ret_far:
  pop R227
  ret

func0:
  push R227
  ld R227,#p$f$func0
  jx f$func0

func1:
  push R227
  ld R227,#p$f$func1
  jx  f$func1
```

As you can see from this example, the main program calls func0, which is mapped in dynamic bank 0, but the call is redirected automatically by the compiler to a different **func0** in the static bank, which will save the current bank number, load the new bank number where func0 actually resides into the bankswitch register, and then jump to the real func0, whose name has been changed to **f$func0**. This new func0 function is generated automatically by the compiler and is stored in a file called **_static_.s**. This is known as a **stub** function.

Note that the return of the functions is no longer a ret, but a jump to **__ret_far**, which is the macro used to restore the previously saved bank number, before returning to the proper bank. This macro is written in the _static_.s file generated by the compiler. It simply executes **pop R227** and issues a **ret** instruction to exit the function and restore the context.

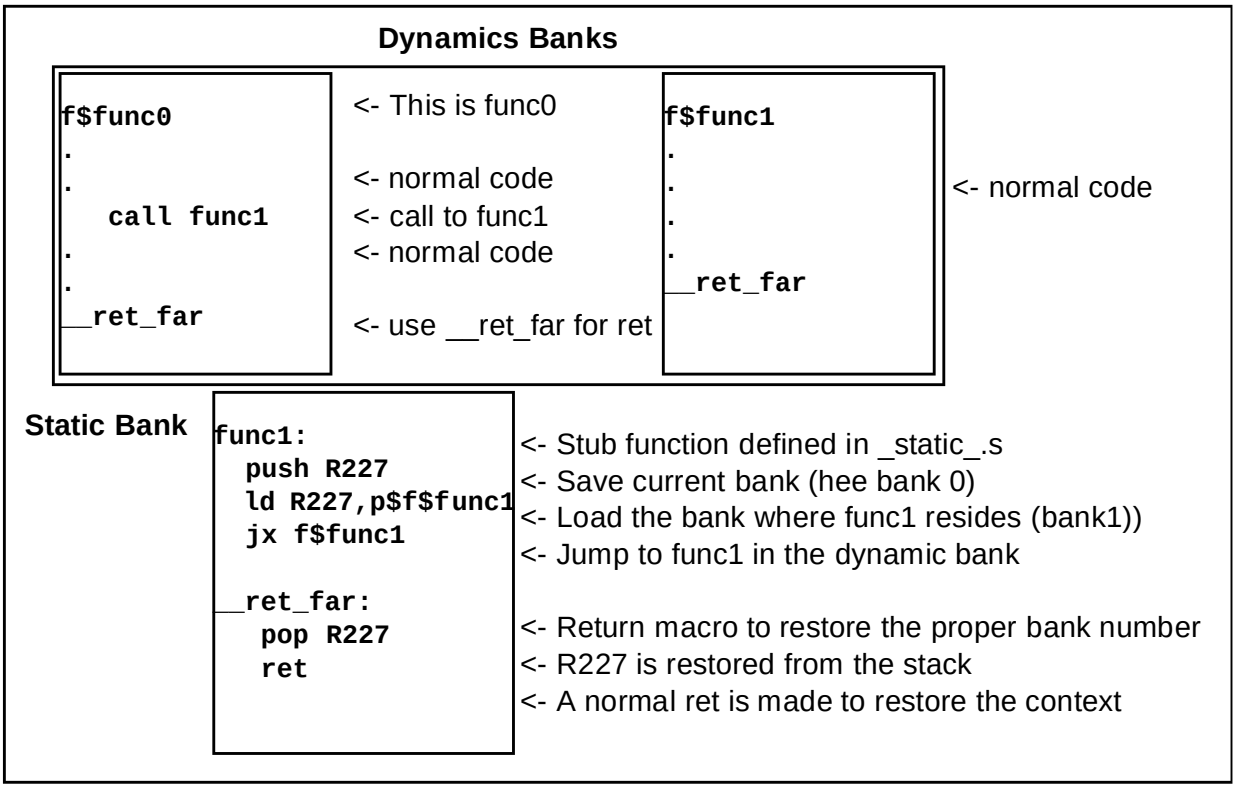
Thus, for each function defined with the **#pragma far** directive, the compiler will generate such a stub function, which automatically manages calls between dynamic banks.

The same process appears here when func1 is called from func0: the call is first directed to the static bank, where the current bank number is saved, then the new bank number is set and a jump to the function in the new bank is performed.

Each function mapped in a dynamic bank, and which can potentially be called from another bank, must be declared using the #pragma far directive.

Note: if a function is mapped in a dynamic bank but is only used internally within the bank (no call to this function from outside this bank), the function **need not** be declared using the #pragma far directive. No stub function will therefore be generated, with consequent speed and space benefits.

Considering only the example of a call between 2 functions mapped in different dynamic banks, the procedure to be followed may be described as follows:



In brief: The only thing that needs to be done by the programmer is to use the #pragma far directive, and to link the _static_.s file with the application. Normally no attention needs to be paid to far function management by the compiler.

The programmer should remember that each time a #pragma far directive is found, the compiler will generate a stub function in the static bank, which requires 11 bytes of memory. It follows, therefore, that the program should be designed so as to keep calls between dynamic banks to a minimum. To put this into proportion, even if 1000 functions have been declared using the #pragma far directive, only 11,000 bytes will be reserved for the stub functions, which amounts to no more than a third of the static bank size.

Speed and code size are significantly optimised, compared to a linear memory where 24-bit addresses are required. The only price to be paid is represented by 11 bytes for each function declared using the #pragma far directive.

2.2 Compiling and linking the example

These files are compiled with:

```
GCC9 -c  bank0.c -o bank0.o      <- The compiler will automatically
                                    generate a _static_.s file
GCC9 -c  bank1.c -o bank1.c      <- to manage far calls.
GCC9 -c main.c  -o main.o        <- It is then necessary to compile and
                                    link this file.
                                     |_
GCC9 -c _static_.s -o _static_.o <- compile the _static_.s file
```

The application is then linked using the appli.scr script file (compulsory for bank switching):

```
GCC9 -m -T appli.scr  _static_.o bank0.o bank1.0 main.o
```

The script file is necessary to provide proper mapping for the linker. In this example, we want to map main.c in the static page (this is compulsory), bank0.c in dynamic bank 0 and bank1.c in dynamic bank 1.

Warning: Be sure to compile and link the _static_.asm file generated by the compiler. Do not insert code into this file, as it may be overwritten during compilation.

With GNU C it is necessary to map all the contents of a file into one specific bank, or several files into the same bank. It is impossible to split the contents of a file among several banks.

2.3 Using a script file

A script file is the memory mapping description file. The linker uses this file to link the application.

Remember that a script file is compulsory for bankswitching.

The script file is structured as follows:

```
OUTPUT_FORMAT(”a.out- st9”)      <- Standard ST9 format
OUTPUT_ARCH(st9)
INPUT(_static_.o main.o bank0.o bank1.o)
                                 <- list here the application object
                                    files
OUTPUT(appli.u)<- indicate the executable

MEMORY {
text :  ORIGIN = p:0000, LENGTH = 32K
                                 <- text is the code for main.c (static
                                    bank)
data :  ORIGIN = d:0000, LENGTH = 32K
                                 <- data is the data space static bank
progsp0 : ORIGIN = p:00:8000, LENGTH = 32K
                                 <- progsp0: code of bank0.c (dyn.
                                    bank0)
```

```
progsp1 : ORIGIN = p:01:8000, LENGTH = 32K
                                <- progsp1: code of bank1.c (dyn.
                                   bank1)
}
LINKED_OBJECT_HAS_BAN KS = 1;   <- this line is compulsory with banks-
                                   witch
                                <- it tells the debugger to use the
                                   bankswitch

SECTIONS {                      <- define here your sections
 progsp0.bk9 : {                <- define here that the .text (code)
                                   of bank0.c will be
  bank0.o(.text)                <- mapped in progsp0. Progsp0 will go
                                   into progsp0.bk9
 {> progsp0                     <- progsp0.bk9 is loaded in the bank 0
                                   by WGDB9.

 progsp1.bk9 : {                <- define here that the .text (code)
                                   of bank1.c will be
  bank1.o(.text)                <- mapped in progsp1
 {> progsp1                     <- define the stack size
 _stack_size = DEFINED(_stack_size) ? _stack_size : 256;
 _user_stack_size = DEFINED(_user_stack_size) ?_user_stack_size : 256;

 .data :{                       <- define the .data section in data
                                   memory space ”data”
    _data_start = .;            <- for all files
    *(.data)
    _data_end = .;

} >data

 .text : {                      <- define the .text section in program
                                   memory ”text”
  _text_start = .;
  *(.text)
  _etext = .;
 DO_OPTION_I;                   <- to make the copy of initial values
                                   from program to
  _text_end = .;                <- data space at startup
 } >text

 .bss : {
  _bss_start = .;               <- define the .bss section
  *(.bss COMMON)
  _ebss =.;
  _bss_end = .;
  _stack_start = DEFINED(_stack_start) ?
                                _stack_start : .;
```

```
    _stack_end = _stack_start + _stack_size;
    _user_stack_start = DEFINED(_user_stack_start) ?
                                    _user_stack_start : _stack_end;
    _user_stack_end = _user_stack_start + _user_stack_size;
  } >data
}
```

Note here that the text in bold shows the names that are completely defined by the user.

Linkage will create 4 files:

– progsp0.bk9 contains the executable to be loaded in dynamic memory bank 0
– progsp1.bk9, contains the executable to be loaded in dynamic memory bank 1
– appli.u, contains the executable code needed by the debugger
– appli.bl9, is a command file needed by the debugger to load progsp0 and progsp1, when loading the application in the debug phase.

2.4 Debugging the application

A debug session with WGDB9 using the bankswitch, needs 3 main files:

– Hardware.gdb
– Appli.u
– Appli.gdb

These 3 files are loaded concecutively when loading the application under the WGDB9 GNU Debugger.

2.4.1 Hardware.gdb

Harware.gdb is the command file needed by WGDB9 to provide the emulator setup. When loading the application executable file (appli.u) at the beginning of a debug session, hardware.gdb is the first file automatically loaded and executed before loading the executable (appli.u), in order to configurate the emulator.

Warning: If no hardware.gdb file is found in the application directory, the debugger will set the default configuration for the emulator, which in our case does not suit the bankswitch application.

It will then be necessary to create a new hardware.gdb file, which must be stored in the application directory.

The hardware.gdb file wich follows is used to debug this example: it should be noticed that this file is totally application dependent. In particular, if external memory is used, it may be necessary to add commands to set the memory extension card properly. In our case we use a memory test board, which needs to be configured before the debug session. These commands are set in the hardware.gdb file given below.

Depending on the type of emulator used (either HDS, or ST9xxx new generation), the commands passed to the emulator will differ. To allow both to be used, a special command, which is always valid, will allow the debugger to recognize which emulator is being configured. This command is:

– "IFTARGET SDBST9" for the HDS emulator

– "IFTARGET SDB9xxx" for the ST9xxx emulator

The configuration for both emulator types is given below:

First case: the HDS emulator is used

```
#********************* ******************************
#If the ST9 Emulator is an HDS model, the debugger
#will automatically use the following target SDBST9:

# set general ST9 parameters
sdb set clock 24<- set CPU clock to 24MHz
# Very important if you use only external memory
sdb set cpu romless             <- simulate romless device,
                                <- because we use the application
                                <- memory only,

!!!! This will give a warning when loading the hardware.gdb file, this
is normal !!!!!!

# set general emulator parameters
sdb set P6 address              <- use non-multiplexed mode
sdb set dm on                   <- program/data enable
sdb set wpmf on                 <- protect program space
sdb set nemf on                 <- protect from wrong memory
                                <- addressing

# Set the bankswitch mode disable even if you use it
# The banks are entirely set under the GDB9 layer
sdb set bswmode disable         <- bankswitch is managed by GDB9
                                <- and not by SDBST9
                                <- so it needs to be desabled

# enable the low nibble of port 2
sdb xport 4 1<- this is the command to tell the
                                <- emulator that the port2 low
                                <- nibble is used for bankswitch,
                                <- this is equivalent to BSL_EN1 =
                                <-1 on the device
# enable the high nibble of port 2
                                <- this is the command to tell the
sdb xport 5 1                   <- emulator that the port2 high
                                <- nibble is used for bankswitch,
                                <- this is equivalent to BSH_EN1 =
                                <- 1 on the device
```

```
# set ST90R91 mapping           <- use only external memory
sdb map 0 7fff swe              <- on the test board
sdb map 8000 ffff uwe
sdb map /0 /7fff swe
sdb map /8000 /ffff uwe

# Don't forget the RESET to validate the status of
# modifications realised above
sdb reset                       <- very important

# Set the memory test board to P/D on P5.3
                                <- particular to our memory
                                <- extension board
# Set segment 08 to 6Fh         <- we write 2 segments (registers)
sdb sr 0xE3 0x08                <- to select the right P/D pin
sdb sm 0x8000 0x6F
# Set segment 09 to EBh
sdb sr 0xE3 0x09
sdb sm 0x8000 0xEB
# Set the return bank to 00
sdb sr 0xE3 0x00

# set the ST9 P/D signal on P5.3
 <- set P5.3 as alternate function to
set R234=3<<2                 <- use it as P/D
set R244=R244|(1<<3)
set R245=R245|(1<<3)
set R246=R246&(~(1<<3))

endiftarget
#end iftarget SDBST9
#******************** ***************
```

Second case: The ST9xxx emulator is used

```
#******************** ****************************
#If ST9 Emulator is an ST9xxx model then the debugger
#will automatically use the following target SDB9XXX:

iftarget SDB9XXX

  bankswitch on

  pd_signal used
  map p:0 0x7FFF sw
  map p:1:0x8000 0xFFFF sr
  map p:0:0x8000 0xFFFF sr

  map d:0 0x7FFF sw
```

```
  map d:0:0x8000 0xFFFF sw
  map d:1:0x8000 0xFFFF sw

endiftarget
#end iftarget SDB9XXX
#******************** ***************
```

2.4.2 Appli.u

Appli.u is the executable file created following successfull linkage; it is loaded directly after hardware.gdb and contains the code mapped in the static bank (program and data), but not the code mapped into the dynamic banks. The code mapped into the dynamic banks will be loaded afterwards, with the appli.gdb file.

2.4.3 Appli.gdb

Appli.gdb is a command file specific to the application, it is loaded after appli.u (the executable), and consequently after hardware.gdb.

When using the bankswitch, this file is compulsory if code or data needs to be mapped in dynamic banks.

The main purpose of this file is to configure the debugger for a debug session with special application settings.

When using the bankswitch, this file is also used to load the code into the memory banks. This is done by making a source of the appli.bl9 file, which downloads the code into the external memory banks.

As the executable has already been loaded, it is then possible to use arguments specific to the application, known to the debugger.

The very simplest minimal appli.gdb file is:

```
# load bank modules (generated by script file)
# If you edit this file appli.bl9, you will see it
# contains instructions to download all files with
# extension .bk9 created by the linker.

source appli.bl9 <- the only command given to WGDB9, this load a
                 <- command  file, executed by the debugger:
    appli.bl9    # This file was automatically generated by the linker.
    =========    # DO NOT EDIT.
                 dlbank progsp0.bk9     <- download progsp0.bk9 file
                 dlbank progsp1.bk9     <- download progsp1.bk9 file
```

As the appli.gdb file is loaded after the appli.u file, the labels and parameters particular to the application are known by the debugger and they can now be used for application settings.

For example, if we want to reset all the . bss section to 0, it is possible to insert the following in appli.gdb:

```
# Reset bank bss because will not be reset by c-start
set R226=0                          <- set the data bank register to 0
                                    <- and fill the memory with 00
sdb fm /$_bss_bank0_start /$_bss_bank0_end 00

set R226=1                          <- set the data bank register to 0
                                    <- and fill the memory with 00
sdb fm /$_bss_bank1_start /$_bss_bank1_end 00
```

3 HARWARE CONSIDERATIONS

If we now consider the device itself, when bankswitching is used A15 will select whether the common bank 0 (static bank, from 0000h to 7FFFh) is addressed, or if one of the other 256 banks (dynamic banks, from 8000h to FFFFh) is addressed.

The 8 bits of the Bankswitch port (port2) are used as the A23-A16 address bits. This port takes different values on reset, depending on the micro:

– BS Port = FEh whenever A15 = ”0” on the ST90R50
– BS Port = 0Eh whenever A15 = ”0” on the ST90R51/91

The P/D pin, programmed as an alternate function, is used to select either Program or Data space.

BS_PSR = R227 will be used as the program dynamic bank if program space is selected.

BS_DSR = P2DR = R226 will be used as the data dynamic bank if data space is selected.

The bankswitch port logic allows either all 8 bits of port 2 to be used to address external memory, or the low nibble for addressing external memory and the high nibble for application I/O purposes. This choice is made by latching the state of the BSH_EN1 and BSL_EN1 input pins. The reset value of the port will also be modified according to the BSH_EN1 and BSL_EN1 values, given by the following table:

BSH_EN1	BSL_EN1	BS Port Nibble		BS Port reset value			
				R50		R51/91	
		High	Low	Prog	Data	Prog	Data
0	0	I/O	I/O	FFh	FFh	FFh	FFh
0	1	I/O	BS	FEh	FEh	FEh	FDh
1	0	I/O	BS	FEh	FEh	FEh	FDh
1	1	BS	BS	FEh	FEh	0Eh	0Dh

On the emulator, BSH_EN1 and BSL_EN1 may be set by software, using the following commands:

```
- set xport 4 1 <=> BSL_EN1 = 1orset xport 4 0 <=> BSL_EN1 = 0
- set xport 5 1 <=> BSH_EN1 = 1orset xport 5 0 <=> BSH_EN1 = 0
```

For example, if normal bankswitch mode is selected (all 8 bits of port 2 are used as addresses), the following commands will be in hardware.gdb:

```
- set xport 4 1
- set xport 5 1
```

More detailed information can be found on pages 24/178-26/178 of the **ST90R5X Databook**.

4 DESCRIPTION OF THE APPLICATION

This example shows a simplified application using 2x2 external memory banks (2 data banks and 2 program banks). Its purpose is to illustrate the bankswitch mechanism, as well as various memory configurations which can help the user to customise memory mapping.

In this application, the 8 bits of the bankswitch port are used as addresses.

The application uses 3 different C files mapped in 2 different banks. Each file declares variables and functions which may call functions in other banks.

Module List::

– **page.c:** the main program, mapped in the static bank
– **page0.c**: a section of code mapped in dynamic bank 0
– **page1.c**: a section of code mapped in dynamic bank 1
– **init.asm**: an assembly init file, to initialise the ST9 port configuration
– **_static_.s**: the automatically generated file for far call managment

The other important files used for the application are:

– **page.scr:** the script file
– **makefile**: the makefile to compile and link the application with the GMAKE utility
– **hardware.gdb**: the emulator configuration file needed by the debugger
– **page.gdb**: the application configuration file needed by the debugger

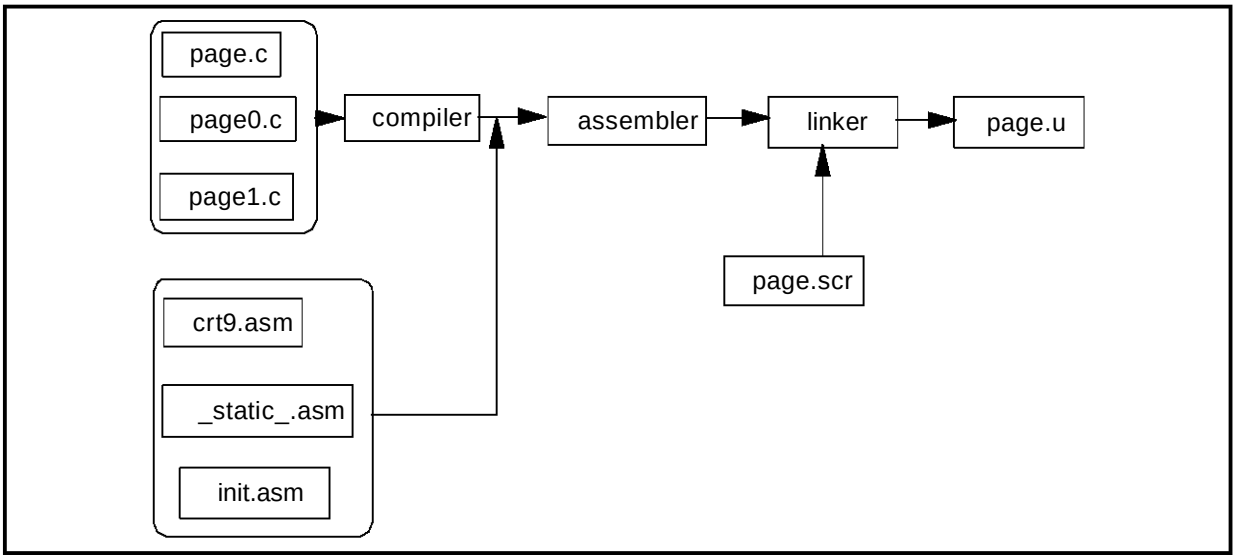
The executable file:

– page.u

The more important generated files:

– **page.bl9**: the command file to download the code and data into the dynamic banks
  . **progsp0.bk9**: the object code of page0.c to be loaded into bank 0 (program space)
  . **progsp1.bk9**: the object code of page1.c to be loaded into bank 1 (program space)
  . **datasp0.bk9**: the data of page0.c to be loaded into bank 0 (data space)
  . **datasp1.bk9**: the data of page1.c to be loaded into bank 1 (data space)

– **page.map**: the map file generated by the linker



4.1 Defining memory mapping with the script file

The script file is used to define the mapping for the application: the program section (.text), the data section (.data or .bss), and the stack parameters.

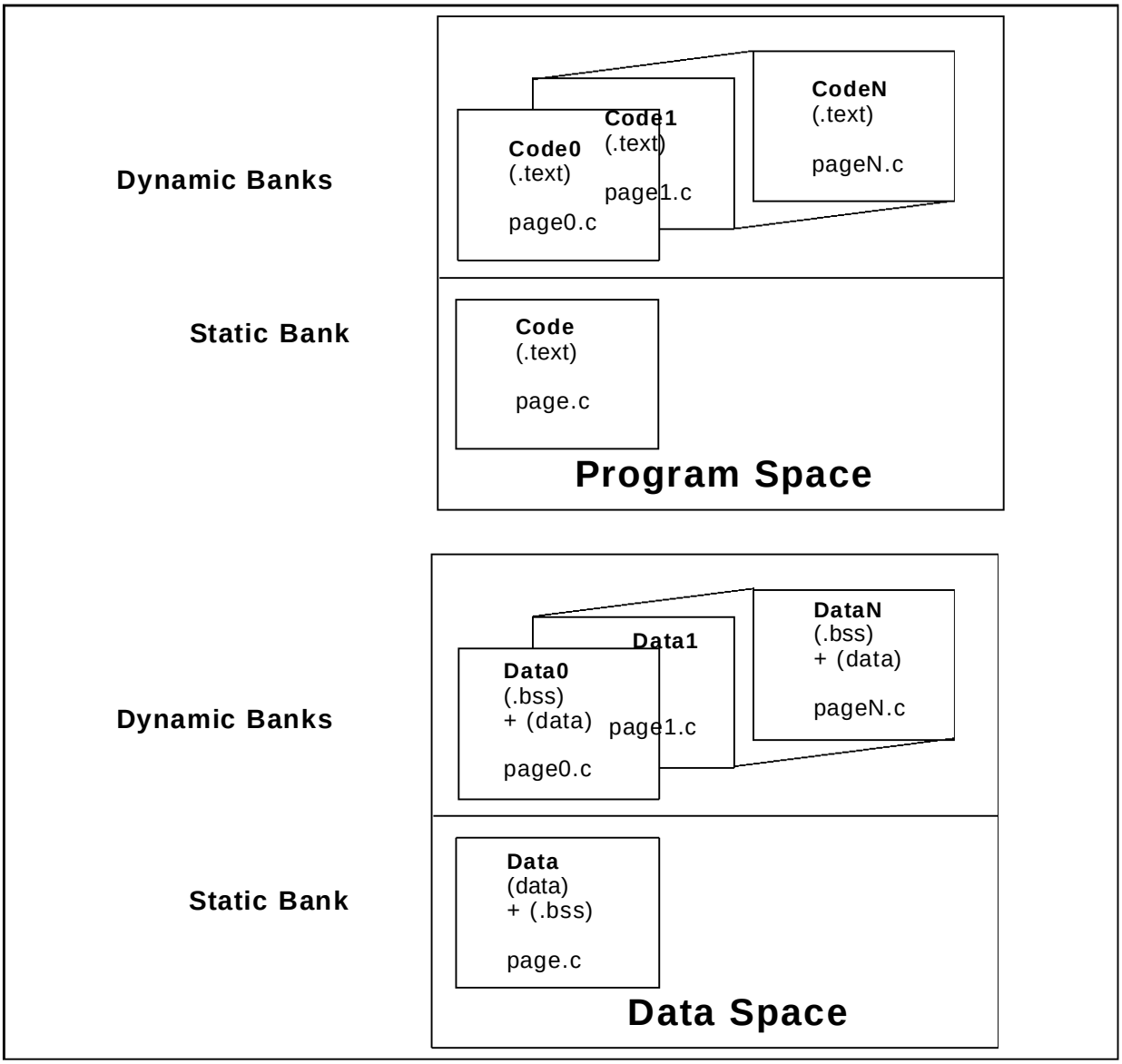
It can also be used to define the input files (object files used in the application) and the startup file. This can however be defined in the makefile.

Two different mappings will be described. The only difference between the two cases is in the script file, which is why for each case the special script file will be described more in depth.

More detailed information can be found on pages 20/40-24/40 of the **LD9** manual.

4.1.1 First mapping: .data section with .bss section in data space

This example maps the code section of each memory bank (static and dynamic) into the corresponding bank in program space. The variables, initialised or not, and the constants (.data and .bss sections) are mapped into the corresponding banks in data space. This is summarised as follows:



The application script file page.scr will be described below:

```
OUTPUT_FORMAT("a.out-st9")
OUTPUT_ARCH(st9)
```

Place all your object files here

```
INPUT(_static_.o page.o page0.o page1.o init.o)
```

Select a startup file. This startup file is not the default, as it does not make a copy of the .data section from program space to data space at startup.

```
STARTUP(crtx9.o)                <- as the default mapping is not used
                                   here, we use a startup file
                                <- which does not make a copy of the
                                   .data section from program
                                <- space to data space at startup
```

Select the name you chose for the application output file.

```
OUTPUT(page.u)
```

Map here your application in terms of hardware resources.

The text, data, progsp0,... are labels used to define the memory mapping.

```
MEMORY {
text : ORIGIN = p:0000, LENGTH = 32K
                                     <- static prog. memory
data : ORIGIN = d:0000, LENGTH = 32K
                                     <- static data memory
progsp0 : ORIGIN = p:00:8000, LENGTH = 32K
                                     <- dyn. progr. memorry, bank 0
datasp0 : ORIGIN = d:00:8000, LENGTH = 32K
                                     <- dyn. data memory, bank 0
progsp1 : ORIGIN = p:01:8000, LENGTH = 32K
                                     <- dyn. progr. memory, bank 1
datasp1 : ORIGIN = d:01:8000, LENGTH = 32K
                                     <- dyn. data memory bank 1
}
```

p is for program memory, **d** is for data memory

p:01:8000 -> bank 1, in program space, starting at address 8000h

This line declares that banks are used. It is compulsory if bankswitch is used.

```
LINKED_OBJECT_HAS_BANKS = 1;
```

Define here all your sections, this will map each object file in the specific bank using the labels defined above. This is why each file must be within a specific bank.

Warning: A file of more than 32K must be split into several smaller files.

The linker will generate as many files, with extension .bk9, as the number of sections which you have defined.

As you can see below, the .text section is stored in program space, and the .bss and .data sections are stored in data space.

```
SECTIONS {
 progsp0.bk9 : {                <- this put the .text section of
                                   page0.c into progsp0

   _text_bank0_start = .;
   page0.o(.text)
   _text_bank0_end = .;
```

```
} > progsp0

progsp1.bk9 : {                 <- this put the .text section of
                                   page1.c into progsp1
  _text_bank1_start = .;
  page1.o(.text)
  _text_bank1_end = .;
} > progsp1

datasp0.bk9 : {                 <- this maps the .bss and .data sec-
                                   tions of page0.c
  _bss_bank0_start = .;         <- into datasp0
  page0.o(.bss .data)
  _bss_bank0_end = .;
} > datasp0

datasp1.bk9 : {                 <- this maps the .bss and .data sec-
                                   tions of page1.c
  _bss_bank1_start = .;         <- into datasp1
  page1.o(.bss .data)
  _bss_bank1_end = .;
} > datasp1
```

The _data_bank0_end,... arguments can be directly used in your application. In this particular case, they remain unused and could therefore have been omitted.

The default stack size is set to 256. It can also be modified by linking options. The _stack_size argument can be used directly as a variable name in the program.

```
_stack_size = DEFINED(_stack_size) ? _stack_size : 256;
_user_stack_size = DEFINED(_user_stack_size)? _user_stack_size : 256;
```

This part is common to many applications, it defines the static sections.

```
.text : {
   _text_start = .;
   *(.text)
   _etext = .;
   _text_end = .;
}>text
.bss : {
   _bss_start = .;
   *(.bss .data COMMON)
   _ebss = .;
   _bss_end = .;
   _stack_start = DEFINED(_stack_start) ?
                                _stack_start : .;
   _stack_end = _stack_start + _stack_size;
   _user_stack_start = DEFINED(_user_stack_start) ?
                                _user_stack_start : _stack_end;
   _user_stack_end = _user_stack_start + _user_stack_size;
```

```
   } >data
}
```

Warning: As it is not the default, the application must be linked using the **-nol** option.

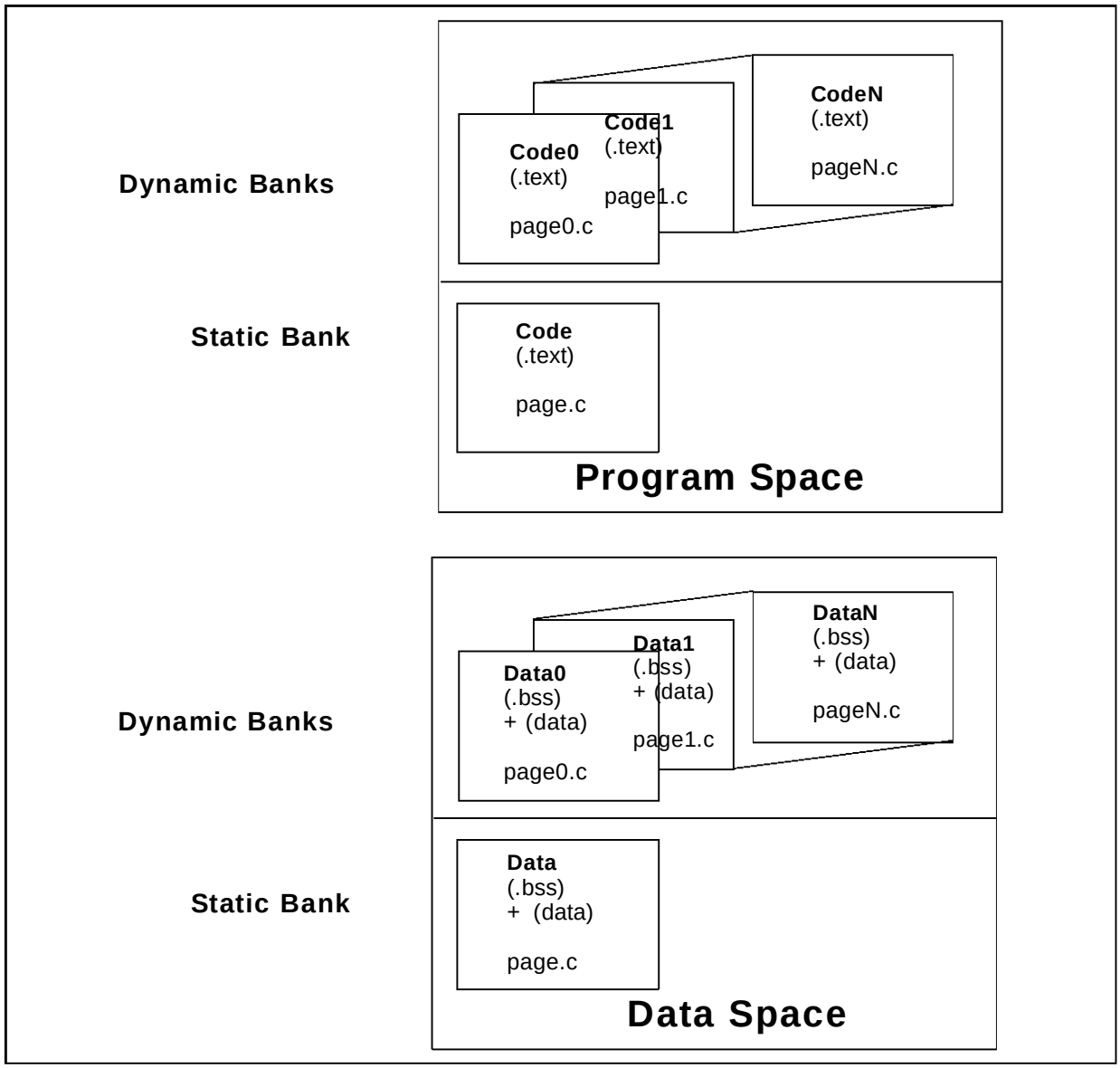
The resulting mapping can be observed from page.map file to check the memory allocation.

Refer to the LD9 manual for further information.

4.1.2 Second mapping: .data section in program space, copied at startup in data space

This example maps the code section of each dynamic memory bank into the corresponding bank in program space. For these dynamic banks, the initialised variables and the constants (.data section) are mapped into the static bank in data space.

This is summarised as follows:



The script file used for this mapping is described below:

```
OUTPUT_FORMAT("a.out-st9")
OUTPUT_ARCH(st9)
INPUT(_static_.o page.o page0.o page1.o init.o)
STARTUP(crt9.o)              <- this is the default startup file
OUTPUT(page.u)
```

```
MEMORY {
text :  ORIGIN = p:0000, LENGTH = 32K
data :  ORIGIN = d:0000, LENGTH = 32K
progsp0 : ORIGIN = p:00:8000, LENGTH = 32K
datasp0 : ORIGIN = d:00:8000, LENGTH = 32K
progsp1 : ORIGIN = p:01:8000, LENGTH = 32K
datasp1 : ORIGIN = d:01:8000, LENGTH = 32K
}
LINKED_OBJECT_HAS_BANKS = 1;     <- remember this line is compulsory

SECTIONS {
 progsp0.bk9 : {                 <- .text section of page0.c in progsp0
  _text_bank0_start = .;
  page0.o(.text)
  _text_bank0_end = .;
 } > progsp0

 progsp1.bk9 : {                 <- .text section of page1.c in progsp1
  _text_bank1_start = .;
  page1.o(.text)
  _text_bank1_end = .;
 } > progsp1

 datasp0.bk9 : {                 <- .bss section of page0.c in datasp0
  _bss_bank0_start = .;
  page0.o(.bss)
  _bss_bank0_end = .;
 } > datasp0

 datasp1.bk9 : {                 <- .bss section of page1.c in datasp1
  _bss_bank1_start = .;
  page1.o(.bss)
  _bss_bank1_end = .;
 } > datasp1

 _stack_size = DEFINED(_stack_size) ? _stack_size : 256;
 _user_stack_size = DEFINED(_user_stack_size)? _user_stack_size : 256;

 .text : {
  _text_start = .;
  *(.text)
  _text_end = .;
  DO_OPTION_I;                   <- specify that copy of .data section
                                    from program
                                 <- space to data space at startup will
                                    appen
  _etext = .;
```

```
  } >text

  .bss : {                          <- data will contain all .data sec-
                                       tions and .bss of
                                    <- page.c
    _bss_start = .;
    *(.bss .data COMMON)
    _ebss = .;
    _bss_end = .;
    _stack_start = DEFINED(_stack_start) ?
                                    _stack_start : .;
    _stack_end = _stack_start + _stack_size;
    _user_stack_start = DEFINED(_user_stack_start) ?
                                    _user_stack_start : _stack_end;
    _user_stack_end = _user_stack_start + _user_stack_size;
  }>data
}
```

5 SETTING UP THE EMULATOR WITH WGBD9

In order to perform the emulator setup, some information must first be passed to WGDB9. This information is mostly contained in two files, hardware.gdb and page.gdb (same name as the application).

The order in which these files are loaded is very important: the harwdare.gdb file must be loaded first, followed by page.u and then by the page.gdb file.

More detailed information can be found in the **WGDB9** manual.

Note: General emulator settings should be contained in the hardware.gdb file, while specific emulator settings should be contained in the application.gdb file. As the application.gdb file is loaded after the application.u file, it is possible to use the application's context (variables, labels, etc.).

These files are very important and an error at this point could be fatal or hide problems during debug ,even if there is no error message.

5.1 The Hardware.gdb file

```
#********************************************
#If ST9 Emulator is an HDS model then the debugger
#will automatically use the following target SDBST9:

iftarget SDBST9
  # set general ST9 parameters
  sdb set clock 24
  sdb set cpu romless
  # set general emulator parameters
  sdb set P6 address
  sdb set dm on
  sdb set wpmf off
  sdb set nemf on
  sdb set bswmode disable
  #sdb set adctrl extended

  #Set the port 2 as bankswitch port (8 bits)
  sdb xport 4 1
  sdb xport 5 1

  # set ST90R91 mapping
  sdb map 0 7fff uwe
  sdb map 8000 ffff uwe
  sdb map /0 /7fff uwe
  sdb map /8000 /ffff uwe

  # reset
  sdb reset
```

```
  # set port 5 bit 3 as alternate function PD
  #set R234=3<<2
  #set R244=0x08
  #set R245=0x08
  #set R246=0x00
  sdb sr ea 3<<2
  sdb sr f4 08
  sdb sr f5 08
  sdb sr f6 00


  # validate banks on test board
  #set R227=0x08
  #set (char *)*0x8000=0x6f
  #set R227=0x09
  #set (char *)*0x8000=0xeb
  sdb sr e3 08
  sdb sm 0x8000 0x6f
  sdb sr e3 09
  sdb sm 0x8000 0xeb

  # Set the bank 0 for default bank
  set RR226=0

endiftarget
#end iftarget SDBST9
#******************** ***************

#******************** ******************************
#If ST9 Emulator is an ST9xxx model then the debugger
#will automatically use the following target SDB9XXX:

iftarget SDB9XXX

  bankswitch on

  pd_signal used
  map p:0 0x7FFF sw
  map p:1:0x8000 0xFFFF sr
  map p:0:0x8000 0xFFFF sr
  map d:0 0x7FFF sw
  map d:0:0x8000 0xFFFF sw
  map d:1:0x8000 0xFFFF sw

endiftarget
#end iftarget DB9XXX#************************** ****
```

5.2 The page.gdb file

It is necessary to define a file with the same name as the application name, with a .gdb extension (here page.gdb). This file is used to load the application settings.

In our case we use the minimum settings, which correspond to:

```
# load bank modules (generated by script file)
# If you edit this file page.bl9, you will see it
# contains instructions to download all files with
# extension .bk9 created by the linker.
source page.bl9
```

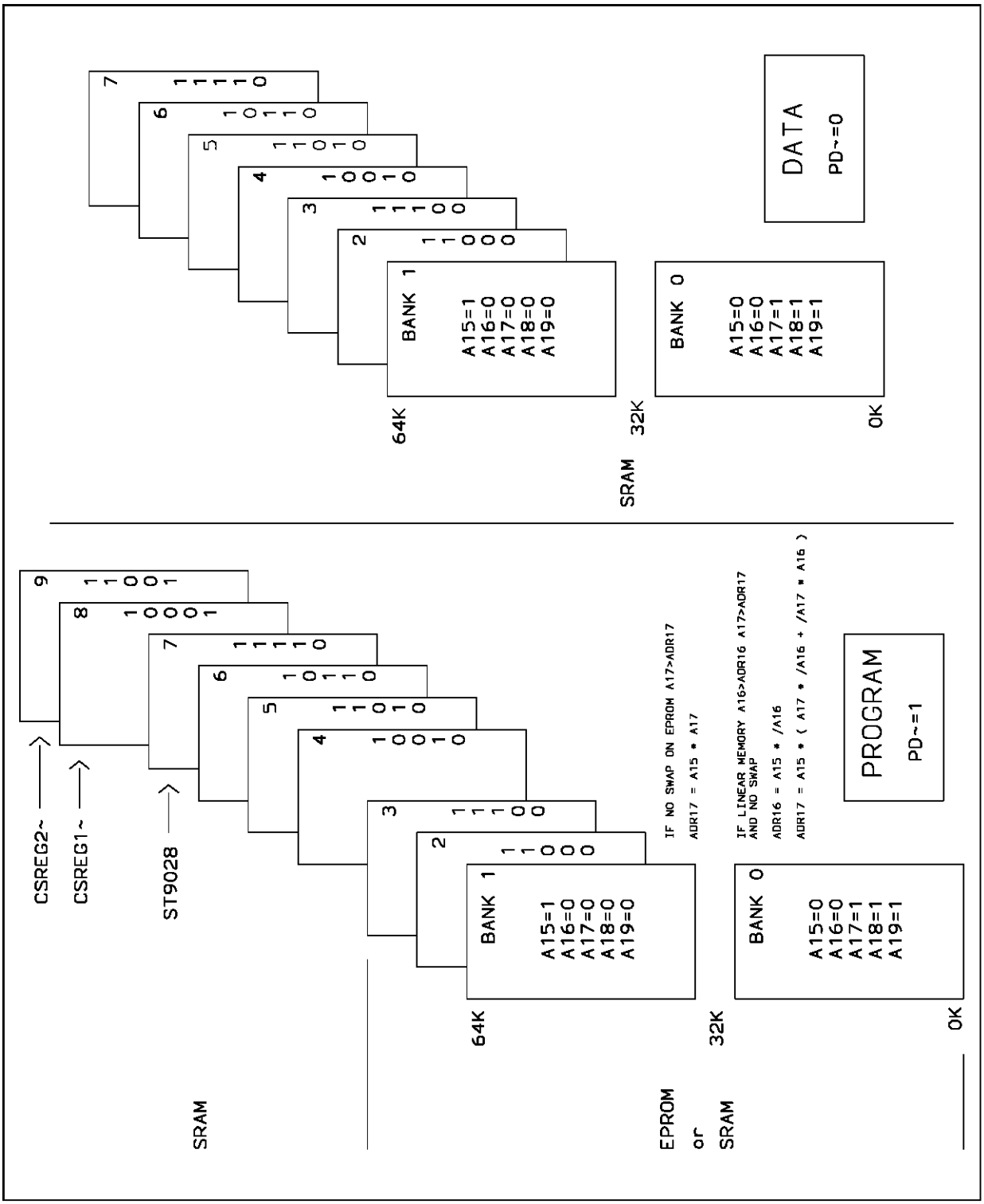
5.3 The memory test board

The memory test board used can be specific to each application. In this application we used the ST90R5X TESTBOARD MB087.

This test board contains 2x128 Kbyte memory blocs for program space and 2x128 Kbytes memory blocs for data space. Two segments, virtually placed in program space bank 8 and 9, are used to select various options on the board. In our application the P/D pin is selected on port 5 bit 3, by setting segment 8 to 6Fh and segment 9 to EBh (this enables multiplexers to select P5.3).

This is done in the hardware.gdb file, when loading the page.u application using WGDB9.

Figure 2. ST90R5x MB087 Memory Test Board



6 APPLICATION FILE LISTINGS

6.1 The page.c file:

```
 /*************************************************************/
/* File:                    PAGE.C                               */
/* Program bank location:   STATIC BANK                          */
/* Data bank location:      STATIC BANK                          */
/*                                                               */
/* Comments: This is the main program, it defines global variables */
/*           and call functions in dynamic banks                 */
/*************************************************************/
#include<bank.h>

#include ”global.h”   /* Include global variables       */

extern void Func0();  /* External functions used in main  */
extern void Func1();

void Func();

static char say_it[] = ”Static page”;

void main(void) {

 Func();              /* Call to functions in static page normally */

 value = 0 ;          /* Global variables can be accessed normally */
                      /* as they are in static page */

 Func0();             /* Call Func0 in bank 0 */
                      /* The bank switch is manage automatically */
                      /* for program space only, as Func0 was    */
                      /* declared as #pragma far (func0) only    */

 value = (int) R_DATA_BANK;
                      /* Set value to data bank number, just to */
                      /* visualise it */

 Func1();             /* Call Func1 in bank 1 */
                      /* The bank switch is manage automatically */
                      /* for program space and data space, as    */
                      /* Func0 was declared as                   */
                      /* #pragma far (func0) only                */

 value = (int) R_PROG_BANK;
                      /* Set value to program bank number, just to */
                      /* visualise it */
```

```
 value = RR_BANK;
                    /* Set value to bank register (RR226), just to */
                    /* visualise it */
};
 /*************************************************************/
/*                                                             */
/*  Define any function you like in static bank and use it normally  */
/*                                                             */
 /*************************************************************/
void Func() {       /* Include here the code you need          */
                    /* It will be located in static page       */
 asm(”ldw RR20, RR22”);
};
```

6.2 The page0.c file:

```
 /*************************************************************/
/* File:                    PAGE0.C                            */
/* Program bank location:   DYNAMIC BANK   0                   */
/* Data bank location:       DYNAMIC BANK   0                  */
/*                                                             */
/* Comments: This file shows how to declare a function in the  */
/*           dynamic page 0. It also makes a call to Func1     */
/*           located in dynamic page 1                         */
 /*************************************************************/

#include<bank.h>

char var0;          /* These variables will be located in the */
char bank0_variable;  /* dynamic page 0                       */
const char cst0 = 11;
char say_it0[] = ”THIS IS BANK 0”;

extern void Func1();
extern int global_var; /* Global variable in static page   */

#pragma far(Func0)    /* Here you declare Func0 as far        */
                      /* to tell the compiler Func0 will be   */
                      /* called from other banks              */
void Func0() {
 asm(”nop”);
 var0 = 12;
 Func();              /* Call a function in static page normally  */
 var0 = global_var;   /* It is possible to access and modify */
                      /* variables in static and dynamic pages    */
 Func1();             /* Call a function indynamic page normally  */
 bank0_variable = 0xFF;
};
```

6.3 The page1.c file:

```
 /*************************************************************/
/* File:                    PAGE1.C                            */
/* Program bank location:   DYNAMIC BANK 1                     */
/* Data bank location:      DYNAMIC BANK 1                     */
/*                                                             */
/* Comments:  This file shows how to declare a far function taking */
/*            also consideration of the data bank register     */
/*                                                             */
 /*************************************************************/

#include<bank.h>

char bank1_variable;
char save_bank;

static char say_it1[] = ”THIS IS BANK 1”;
                      /* Define here Func1 as far */

#pragma far(Func1,p$bank1_variab le)
                      /* The bank where bank1_variable */
                      /* was mapped is also manage   */
                      /* A call to Func1 will then   */
                      /* set the prog and data bank  */
                      /* registers */

/* You can also define: #pragma far(Func1,1) ( where 1 is the data
   bank number)      */

EXTERN_IN_BANK(bank0_ variable);
                      /* Use this definition to make */
EXTERN_IN_BANK(bank1_ variable);
                      /* GNU recognise the p$name    */
                      /* symbol as the data bank     */
                      /* BANK_OF(name) will then give*/
                      /* the data bank number        */
extern char bank0_variable;

void Func1_local();

void Func1() {

 bank1_variable = 0x12;
                      /* Do anything you need with the */
                      /* variables defined in bank 1   */

                      /* If you want do use variables  */
                      /* in another bank, first :      */
                      /* Change the data bank to the   */
```

```
                        /* right one */
  R_DATA_BANK = BANK_OF(bank0_variable);
                        /* This will set the data bank to */
                        /* one where bank0_variable is    */
                        /* located                        *:

  bank0_variable = 0xAA;
                        /* Perform the operation you want*/
                        /* in this particular bank       */

  R_DATA_BANK = BANK_OF(bank1_variable);
                        /* Restore the data bank register */
                        /* to the actual bank             */

  bank1_variable = 0x56;
                        /* Do anything you need with the */
                        /* variables defined in bank 1   */

  Func1_local();        /* Use local function declared   */
                        /* in the same bank if you need  */

};/* end Func1 */

 /*************************************************/
/*                                                 */
/* Define any local function you like and use it normally   */
/*                                                 */
 /*************************************************/
void Func1_local(void){
                        /* Do not declare this function  */
                        /* as far, because it is used    */
                        /* only in bank1                 */
  bank1_variable = 0;
}
```

6.4 The init.asm file:

```
; This file sets the port configuration for bankswitch
; and the memory test board

        .include ”..\include.st9\st905x.inc”
        .text
;
;#init P0,P1,P6,P2.0,P5.3 for bank switching : c2=0,c1=1,c0=1
;
;******************* ********************************** ;
;****  FUNCTION: init_bank_hardware            *******;
;******************* ********************************** ;

init_bank_hardware::

;#P0C0,1,2 = R240,241,242 Page 2
;
; Alternate function, push-pull, TTL
;
        spp     #P0C_PG              ; set port 0 page
        ld      P0C0R,#0xff
        ld      P0C1R,#0xff
        ld      P0C2R,#0x00

;#P1C0,1,2 = R244,245,246 Page 2
;
; Alternate function, push-pull, TTL
;

        spp     #P1C_PG             ; set port 1 page
        ld      P1C0R,#0xff
        ld      P1C1R,#0xff
        ld      P1C2R,#0x00

;
;#P5C0,1,2 = R244,245,246 Page 3
;
; Alternate function, push-pull, TTL, only for bit 3
;
        spp     #P5C_PG             ; set port 5 page
        or      P5C0R,#8
        or      P5C1R,#8
        and     P5C2R,#(~8)

;#P6C0,1,2 = R248,249,250 Page 3
;
; Alternate function, push-pull, TTL
;
```

```
        spp     #P6C_PG             ; set port 6 page
        ld      P6C0R,#0xff
        ld      P6C1R,#0xff
        ld      P6C2R,#0x00

; Set P/D on P5.3 on memory test board
;                  <=>
; Set segment 08 and 09 in program space

 spm                  ; set program memory space
 ld BS_PSR,#08h Janice; load bank register with 08 to select 8th bank
 ld 8000,#06Fh        ; load 6F in this bank (just a write only regis-
                        ter)
 ld BS_DSR,#09h       ; load bank register with 09 to select 9th bank
 ld 8000,#0EBh        ; load EB in this bank (just a write only regis-
                        ter)
 clr BS_PSR           ; reset the program space bankswitch register
 clr BS_DSR           ; reset the data space bankswitch register
 sdm; reset the data memory space

 jx  end_init         ; end init_bank hardware
```

6.4.1 The CRT9.asm file:

.include ”c:\gnu9\include.st9\st905x.inc”

```
INIT_CIC = 8fh        ; CIC = IT disabled + Nested Mode + CPL = 7
K_INITCLOCKMODE = 20h ; R235 = both stack in memory + clock divided by 2
K_INITWCR = 40h       ; R252 = zero wait state + watchdog enabled

 .text
;;.org    0
 .word   __Reset
 .blkb   50           ; reserve room for the interrupt vectors

 .text
;--------------------------------------------------------
;Reset routine
;
; WARNING : it is important to set rr12 to something NOT zero.
; This is because for the debugger a frame pointer null means
; the function has no parent frame (ie cannot go up). This is OK
; for the main routine, but not for routines called by main.
; If -fomit-frame-pointer is used for compiling, rr12 could not
; be set in main, nor in functions called by main...
; Furthermore, it seems a good idea to initialize the current frame
; pointer to the current stack pointer.
;
;--------------------------------------------------------
```

```
  .global __Reset
__Reset:
  spp  #0                       ; page 0 to access WCR
  ld   R252,#K_INITWCR          ; WCR = zero wait state
  ld   R235,#K_INITCLOCKMODE    ; init CLOCKMODE (both stack in memory)
  ld   R230,#INIT_CIC           ; CIC for our program

  srp  #0                       ; set register pointer to 0
  sdm                           ; set data memory
  ldw  RR238,#_stack_end        ; setup stack
  ldw  RR236,#_user_stack_end   ; setup user stack

  jx   init_bank_hardware       ; Initialize P/D and ports
end_init::

  call _K_InitDataBss           ; init data & bss section
  ei                            ; enable interrupts
  ldw  rr12,RR238               ; make sure rr12 is NOT zero
  call main                     ;
  halt                                  ;
;-------------------------------------------------- ------
;Function to initialize the data, bss and stack sections.
;
; input : none.
;  Data space is selected.
;
; output : none.
;  rr0,rr2,rr4,rr6 are trashed.
;  Data space is selected.
;
; Note : LD9 creates the following symbols :
;
;  _data_start points to start of run time data area,
;  _data_end points to end of run time data area,
;  _bss_start points to start of run time bss area,
;  _bss_end points to end of run time bss area,
;  _text_start points to start of text segment,
;  _text_end points to end of text segment.
;  _stack_start points to start of stack segment,
;  _stack_end points to end of stack segment.
;
; Note :  option I must be used with LD9 in order to get
;         the initialize data information at the end of the text section
          (in ROM).
;
;         In that case '_text_end - (_data_end - _data_start)' gives
;         the start address of the initialized data information in
;         the text segment.
;
```

```
; Note :  be careful to save the return address before clearing the
;        BSS section,because the kernel stack belongs to the BSS
;        section.
;
 ;---------------------------------------- ----------------
  .global __K_InitDataBss
__K_InitDataBss:
  popwrr6                      ; save return address in rr6
;
; Init data area
;
  ldw  rr0,#_data_start        ; start of run-time data area
  ldw  rr4,#_data_end          ; end of run time data area
  subw rr4,rr0                 ; rr4 = length of initialize data
  jrz  no_data                 ; if empty
  ldw  rr2,#_text_end          ; end ROMed data area
  subw rr2,rr4                 ; start of ROMed data area
init_data:
  lddp   (rr0)+,(rr2)+        ; init data section
  dwjnz  rr4,init_data        ;
no_data:
;
; Init bss section
;
  xor    r4,r4                ; r4 = 0.
  ldw    rr0,#_bss_start      ; start of run-time bss area
  ldw    rr2,#_bss_end        ; end of run time bss area
  subw   rr2,rr0              ; rr2 = length of bss area
  jrz    no_bss               ; if bss is empty
init_bss:
  ld     (rr0)+,r4            ; clear all bss section (Data space)
  dwjnz  rr2,init_bss         ;
no_bss:
;
; Init stack section (not really necessary, but cleaner)
;
; (here r4 = 0)
;
  ldw    rr0,#_stack_start    ; start of run-time stack area
  ldw    rr2,#_stack_end      ; end of run time stack area
  subw   rr2,rr0              ; rr2 = length of stack area
  jrz    endinit              ; if stack is empty
init_stack:
  ld     (rr0)+,r4            ; clear all stack section (Data space)
  dwjnz  rr2,init_stack       ;

endinit:
  jp     rr6                  ; return to caller.
```

6.5 The _static_.s file:

```
; This file is automatically generated by GCC9 - Do not edit.
; Compile this file with 'gcc9 -g'.
.ifc ndf __ret_far
__ret_far::
 popR227
 ret
.endc
.ifc ndf __ret_xfar
__ret_xfar::
 popwRR226
 ret
.endc
.ifc ndf __jp_far
   .macro __jp_far funcname
funcname::
 pushR227
 ld R227,#p$f$'funcname
 jx f$'funcname
 .endm
.endc
.ifc ndf __jp_xfar
   .macro __jp_xfar funcname,dataname
funcname::
 pushwRR226
 .word  0xbfe2 ; ldw RR226,#
 .byte  dataname
 .byte  p$f$'funcname
 jx f$'funcname
 .endm
.endc
;File page1.c
 __jp_xfar Func1,p$bank1_variable ; f$Func1
;File page0.c
 __jp_far Func0 ; f$Func0
```

6.6 The makefile:

```
#******************* *****************
#    GNU MAKEFILE
#******************* *****************
# DEFINES :
#**********
IFLAGS = -I..\include
CFLAGS = -c -g -Wa,-alhd -O -fomit-frame-pointer -Wall $(IFLAGS)
ASMFLAGS = -c -g -Wa,-alhd
LDFLAGS = -m -I
```

```
APPLI = page

Clist_SRC = page.c page0.c page1.c
ASMlist_SRC = crt9.asm init.asm _static_.s

# COMMON DEFINES :
#*****************
.SUFFIXES:
.SUFFIXES: .c .asm .st9 .s .s9 .o .scr .u

CC = gcc9
LD = ld9

EXE =  ${APPLI}.u
SCRIPT = ${APPLI}.scr
list_OBJ = $(subst .c,.o,$(filter %.c,$(Clist_SRC)))        \
  $(subst .asm,.o,$(filter %.asm,$(ASMlist_SRC))) \
  $(subst .st9,.o,$(filter %.st9,$(ASMlist_SRC))) \
  $(subst .s,.o,$(filter %.s,$(ASMlist_SRC)))

%.o:%.c
  $(CC) $(CFLAGS) $< -o $@
%.o:%.asm
  $(CC) $(ASMFLAGS) $< -o $@
%.o:%.st9
  $(CC) $(ASMFLAGS) $< -o $@
%.o:%.s
  $(CC) $(ASMFLAGS) $< -o $@

$(EXE) : $(list_OBJ)
  $(LD) $(LDFLAGS) -T $(SCRIPT) -o $(EXE)

# CREATE THE OBJECT FILES :
#**************************
include makedep

.PHONY:_static_.s
_static_.o: _static_.s $(Clist_SRC)

# a target to delete all generated file :
#***************************************
.PHONY:clean
clean:
  del *.s
  del *.o
  del *.l
  del *.u
  del *.bk9
  del *.bl9
```

```
 del *.map
 $(CC) -MM $(IFLAGS) $(Clist_SRC) >makedep


# a target to generate dependencies in makedep file :
# if makedep do not exist, or if is not a regular depency file,
# use gmake -k dep
#******************** ********************************** ******
makedep: $(Clist_SRC)
 $(CC) -MM $(IFLAGS) $(Clist_SRC) >makedep
```

6.7 The page.scr script file (for mapping 1)

```
OUTPUT_FORMAT(”a.out- st9”)
OUTPUT_ARCH(st9)
INPUT(_static_.o page.o page0.o page1.o init.o)
STARTUP(crtx9.o)
OUTPUT(page.u)

MEMORY {
text :  ORIGIN = p:0000, LENGTH = 32K
data :  ORIGIN = d:0000, LENGTH = 32K
progsp0 : ORIGIN = p:00:8000, LENGTH = 32K
datasp0 : ORIGIN = d:00:8000, LENGTH = 32K
progsp1 : ORIGIN = p:01:8000, LENGTH = 32K
datasp1 : ORIGIN = d:01:8000, LENGTH = 32K
}
LINKED_OBJECT_HAS_BAN KS = 1;

SECTIONS {
 progsp0.bk9 : {
  _text_bank0_start = .;
  page0.o(.text)
  _text_bank0_end = .;
 } > progsp0

 progsp1.bk9 : {
  _text_bank1_start = .;
  page1.o(.text)
  _text_bank1_end = .;
 } > progsp1

 datasp0.bk9 : {
  _bss_bank0_start = .;
  page0.o(.bss .data)
  _bss_bank0_end = .;
 } > datasp0

 datasp1.bk9 : {
```

```
    _bss_bank1_start = .;
    page1.o(.bss .data)
    _bss_bank1_end = .;
  } > datasp1

  _stack_size = DEFINED(_stack_size) ? _stack_size : 256;
  _user_stack_size = DEFINED(_user_stack_size)? _user_stack_size : 256;

  .data :{
    _data_start = .;
    *(.data)
    _data_end = .;
    } >data

  .text : {
    _text_start = .;
    *(.text)
    _etext = .;
    _text_end = .;
  } >text

  .bss : {

    _bss_start = .;
    *(.bss COMMON)
    _ebss =.;
    _bss_end = .;
    _stack_start = DEFINED(_stack_start) ?
      _stack_start : .;
    _stack_end = _stack_start + _stack_size;
    _user_stack_start = DEFINED(_user_stack_start) ?
      _user_stack_start : _stack_end;
    _user_stack_end = _user_stack_start + _user_stack_size;

  } >data
}
```

6.8 The page.scr script file (for mapping 2)

Warning: All .data sections will be mapped in the static page

```
OUTPUT_FORMAT(”a.out- st9”)
OUTPUT_ARCH(st9)
INPUT(_static_.o page.o page0.o page1.o init.o)
STARTUP(crt9.o)
OUTPUT(page.u)

MEMORY {
text :  ORIGIN = p:0000, LENGTH = 32K
data :  ORIGIN = d:0000, LENGTH = 32K
progsp0 : ORIGIN = p:00:8000, LENGTH = 32K
datasp0 : ORIGIN = d:00:8000, LENGTH = 32K
progsp1 : ORIGIN = p:01:8000, LENGTH = 32K
datasp1 : ORIGIN = d:01:8000, LENGTH = 32K
}
LINKED_OBJECT_HAS_BAN KS = 1;

SECTIONS {
 progsp0.bk9 : {
  _text_bank0_start = .;
  page0.o(.text)
  _text_bank0_end = .;
 } > progsp0

 progsp1.bk9 : {
  _text_bank1_start = .;
  page1.o(.text)
  _text_bank1_end = .;
 } > progsp1

 datasp0.bk9 : {
  _bss_bank0_start = .;
  page0.o(.bss)
  _bss_bank0_end = .;
 } > datasp0

 datasp1.bk9 : {
  _bss_bank1_start = .;
  page1.o(.bss)
  _bss_bank1_end = .;
 } > datasp1

 _stack_size = DEFINED(_stack_size) ? _stack_size : 256;
 _user_stack_size = DEFINED(_user_stack_size) ? _user_stack_size : 256;


 .data :{
  _data_start = .;
```

```
  *(.data)
  _data_end = .;
  } >data

.text : {
  _text_start = .;
  *(.text)
  _etext = .;
    DO_OPTION_I;
  _text_end = .;
} >text

.bss : {

  _bss_start = .;
  *(.bss COMMON)
  _ebss =.;
  _bss_end = .;
  _stack_start = DEFINED(_stack_start) ?
                                  _stack_start : .;
  _stack_end = _stack_start + _stack_size;
  _user_stack_start = DEFINED(_user_stack_start) ?
                                  _user_stack_start : _stack_end;
  _user_stack_end = _user_stack_start + _user_stack_size;


} >data
```

7 NIBBLE MODE APPLICATIONS

7.1 Introduction

The nibble mode, is a special feature that can be implemented both on the emulator and on the device. This feature allows the user to use the high order nibble of port 2 (4 MSB) as I/Os, for application purposes, and the low nibble (4 LSB) of port 2 as addresses to address 16 dynamic banks for program space and 16 dynamic banks for data space.

Warning: Some precautions must however be taken by the user when manipulating port 2, since writing to the low nibble may change the bank number and crash the program or use the wrong variable contents, if in data space.

A constraint with GNU C exists, in that the whole byte of port 2 is saved before a call to a function mapped in a dynamic bank, and restored when returning from the function. This is done by the stub functions. Thus if the high nibble is modified in the function, its value will be restored to its previous value when returning from the function and the new value will therefore be lost.

This places a constraint on bankswitch management, since the user must always be aware of the value of the high nibble.

The following application note gives a solution to this problem, freeing the user from this problem.

The example provided uses a modified version of the application software used to demonstrate the bankswitch mecanism.

7.2 Defining new stub functions

First of all, in order to correctly understand the problem concerning the stub functions, these must first be described.

There are 2 kinds of stub functions, depending on the the #pragma far directive given to the compiler.

They are represented in table 1.

As can be seen, the bank number, either for program space or data space, is always saved in its entirety on the stack, and restored when returning from the called function. Thus it is clear that, if the high nibble is modified during the far function, returning from this function will restore the high nibble to the saved value.

Table 1.

Directive	#pragma far (funcname)	#pragma far (funcname, data-bank)
Macro used for the call	.macro __jp_far funcname funcname:: push R227 ld R227, #pf'funcname jx f$'funcname .endm	.macro __jp_xfar funcname funcname:: .word 0xbfe2 .byte dataname .byte pf'funcname jx f$'funcname .endm
Return from the call	__ret_far:: pop R227 ret	__ret_xfar:: pop RR226 ret

A solution is to write the new macros in such manner that only the low nibble is saved before a far call, and of course that only the low nibble will be restored when returning from the far call.

These new stub functions are described in the following table:

Table 2.

Directive	#pragma far (funcname)	#pragma far (funcname, data-bank)
Macro used for the call	.macro __jp_far funcname funcname:: push R227 and R227, #0xF0 or R227,#pf'funcname jx f$'funcname .endm	.macro __jp_xfar funcname funcname:: pushw RR226 and R226, #0xF0 or R226, #dataname and R227, #0xF0 or R227, #pf'funcname jx f$'funcname .endm
Return from the call	*__ret_far::* pop r4 and r4, #0x0F and R227, #0xF0 or R227, r4 ret	*__ret_xfar::* popw rr4 andw rr4, #0x0F0F andw RR226, #0xF0F0 orw RR226, rr4 ret

Since the compiler automatically generates the _static_.s file, which contains the expanded macro calls, it is not possible to write anything in this file.

A simple solution is to include the _static_.s file at the end of the file containing the above code. With this solution, since the _static_.s code is written in a conditional compilation manner, using the .ifndef directive, the stub functions will use the macros defined by the user and not the ones in the _static_.s file.

The user will need to follow the following steps:

– create a file (eg: mystatic.s)
– add the new stub function declarations to this file
– add .include "_static_.s" at the end of the file
– compile and link mystatic.s with the application

A short application will now be described to show that the nibble mode can easily be implemented by following the above procedure.

7.3 Main differences with respect to normal mode

The use of the bankswitch in nibble mode can be compared to the normal mode, by considering only the following differences:

On the emulator the only difference compared to the normal mode is to set xport 5 to 0 within the hardware.gdb file.

On the device, this is equivalent to latching BSH_EN1 end BSL_EN1 to 0 and 1 (or to 1 and 0), respectively.

It is necessary to mask the 4 MSBs of port 2 from WGDB9, this is done within the page.gdb file by setting:

```
set bank_prog_and 0x0F
set bank_data_and 0x0F
```

7.4 The nibble mode application example

This application is a test program to demonstrate how to create an application using the bankswitch nible mode and how to test and debug it using the HDS emulator.

The same memory test board is used as for the previous bankswitch application in normal mode.

This application uses the 4 MSBs of port 2 to switch 4 LEDs on or off, while accessing different dynamic banks. It will be shown that, if a function mapped in a dynamic bank modifies the LED status, the return will not change this status.

The following description illustrates the main differences with respect to the normal mode example.

7.4.1 Module list

The source files:

– **page.c:** the main program, mapped in the static bank
– **page0.c**: a part of code mapped in dynamic bank 0
– **page1.c**: a part of code mapped in dynamic bank 1
– **init.asm**: an assembly init file, to initialise the ST9 port configuration
– **mystatic.st9:** the file defining the macros for the far calls and return. At the end of this, **_static_.s**: the automatically generated file for far call managment is included.

Other important files used for the application (debug):

– **page.scr:** the script file - **not modified -**
– **makefile**: the makefile to compile and link the application - **not modified** -
– **hardware.gdb**: the emulator configuration file needed by the debugger
– **page.gdb**: the application configuration file needed by the debugger

7.4.2 The page.c file

```
/********************* ******************************************* ******/
/* File:                    PAGE.C                                    */
/* Program bank location:   STATIC BANK                                */
/* Data bank location:      STATIC BANK                                */
/*                                                                     */
/* Comments: This is the main program, it defines global variables */
/*           and call functions in dynamic banks                       */
/*                                                                     */
 /***************************************** ***************/

# include<bank.h>
# include ”page.h”         /* Header file for new types */

void Delay(char);          /* Definition of functions */
void Blink_Leds_BS_DSR();
void Set_R_DATA_High_Nibble(unsig ned char);
void Set_R_DATA_Low_Nibble(unsign ed char);
void Set_R_PROG_High_Nibble(unsig ned char);

extern void Func0();       / *External functions used in main */
extern typ_struct Func1();

char say_it[] = ”Static bank”;/* You can visualize it with the dump */
int global_var;
char value = 12;

typ_struct var_static;     /* Declare var_static if type typ_static */
```

```
/*********************************************** *******/
/*                                MAIN PROCEDURE                 */
/*                                                               */
/*Comments:                                                     */
/*This procedure shows a possible use of the high nibble    */
/*of port 2 when using the bankswitch feature               */
 /***************************************** ***************/

void main(void) {

 Blink_Leds_BS_DSR();/* This blinks leds on port2 I/Os  */

 Func0();              /* Call Func0 in bank 0 */
                       /* The bankswitch is managed automatically */
                       /* for program space only, as Func0 was    */
                       /* declared as #pragma far (func0) only    */

 value = (int) R_DATA_BANK;
                       /* Set value to data bank number, just to */
                       /* visualize it */
 var_static = Func1(); /* Call Func1 in bank 1 */
                       /* The bank switch is managed automatically*/
                       /* for program space and data space, as */
                       /* Func0 was declared as         */
                       /* #pragma far (func1,1)         */

 Blink_Leds_BS_DSR();  /* This blinks leds on port2 I/Os*/

 value = (int) R_PROG_BANK;
                       /* Set value to program bank number, just to */
                       /* visualize it */

 value = RR_BANK;      /* Set value to bank register (RR226), just to */
                       /* visualize it */

};/* end MAIN */

 /***************************************** ***************/
/*                                                               */
/*This function sets the high nible of BS_DSR to a given value */
/*                                                               */
 /***************************************** ***************/
void Set_R_DATA_High_Nibble(unsig ned char new_value) {
 new_value &= 0xF0;
 R_DATA_BANK &= 0x0F;
 R_DATA_BANK |= new_value;
 }
 /***************************************** ***************/
/*                                                               */
```

```
/*This function sets the Low  nible of BS_DSR to a given value */
/*                                                              */
 /*************************************** ***************/
void Set_R_DATA_Low_Nibble(unsign ed char new_value) {
 new_value &= 0x0F;
 R_DATA_BANK &= 0xF0;
 R_DATA_BANK |= new_value;
 }
 /*************************************** ***************/
/*                                                              */
/*This function sets the high nible of BS_PSR to a given value */
/*                                                              */
 /*************************************** ***************/
void Set_R_PROG_High_Nibble(unsig ned char  new_value) {
 new_value &= 0xF0;
 R_PROG_BANK &= 0x0F;
 R_PROG_BANK |= new_value;
 }
 /*************************************** ***************/
/*                                                              */
/*This function is just a simple delay                        */
/*                                                              */
 /*************************************** ***************/
void Delay(char time){
 char i;
 for (i=0;i<time;i++);
 }

 /*************************************** ***************/
/*This function blinks 4 Leds on port2 high nibble             */
/*                                                              */
/*Comments: It just set high nibble bits, one after another    */
 /*************************************** ****************/
void Blink_Leds_BS_DSR(){
 char i;
 for (i=0;i<10;i++){
  Set_R_DATA_High_Nib ble(0x10); /* port2.4 = 1               */
   Delay(100);
 Set_R_DATA_High_Nibble(0x20);  /* port2.5 = 1               */
   Delay(100);
 Set_R_DATA_High_Nibble(0x40);  /* port2.6 = 1               */
   Delay(100);
 Set_R_DATA_High_Nibble(0x80);  /* port2.7 = 1               */
   Delay(100);
 Set_R_DATA_High_Nibble(0x00);  /* port2 = 0                 */
 }
}
 /*************************************** ***************/
```

7.4.3 The page0.c file

```
 /*************************************** ***************/
/* File:                    PAGE0.C                                */
/* Program bank location:   DYNAMIC BANK   0                       */
/* Data bank location:      DYNAMIC BANK   0                       */
/*                                                                 */
/* Comments: This file shows how to declare a function in the  */
/*           dynamic page 0. It also makes a call to Func1         */
/*           located in dynamic page 1                             */
/******************* ************************************ ******/
#include<bank.h>
#include ”page.h”

char var0;/* These variables will be located in the */
char bank0_variable;   /* dynamic page 0               */
const char cst0 = 11;
char say_it0[] = ”THIS IS BANK 0”;

typ_struct var0_struct; /* Define var0_struct as a type typ_struct*/

extern global_var;
extern typ_struct Func1();
extern void Blink_Leds_BS_DSR();
extern void Set_R_DATA_High_Nibble(unsigned char);
extern void Set_R_PROG_High_Nibble(unsigned char);

/******************* ************************************ *******/
#pragma far(Func0)     /* Here you declare Func0 as far        */
                       /* to tell the compiler Func0 will be     */
                       /* called from other banks                */
/******************* ************************************ *******/
void Func0(void) {

 asm(”nop”);
 var0 = 12;

 Blink_Leds_BS_DSR();

 Set_R_DATA_High_Nibble(0xAA);
 Set_R_PROG_High_Nibble(0xBB);

 var0 = global_var;    /* It is possible to access and modify    */
                       /* variables in static and dynamic pages  */

 Blink_Leds_BS_DSR();

 Set_R_DATA_High_Nibble(0xCC);
 Set_R_PROG_High_Nibble(0xDD);
```

```
 var0_struct = Func1(); /* Call a function in dynamic page normally*/

 Blink_Leds_BS_DSR();

 bank0_variable = 0xFF;
};
```

7.4.4 The page1.c file

```
 /***************************************** ***************/
/* File:                    PAGE1.C                        */
/* Program bank location:   DYNAMIC BANK 1                 */
/* Data bank location:      DYNAMIC BANK 1                 */
/*                                                         */
/* Comments:  This file shows how to declare a far function taking */
/*           also consideration of the data bank register         */
/*                                                         */
 /***************************************** ***************/

#include<bank.h>

#include ”page.h”

char bank1_variable;
char save_bank;

char say_it1[] = ”THIS IS BANK 1”;

extern void Blink_Leds_BS_DSR();/* External functions      */
extern void Set_R_DATA_High_Nibble(unsigned char);
extern void Set_R_DATA_Low_Nibble(unsigned char);
extern void Set_R_PROG_High_Nibble(unsigned char);

extern char bank0_variable;
                              /* Define here Func1 as far     */
#pragma far(Func1,p$bank1_variab le)
                              /* The bank where bank1_variable*/
                              /* was mapped is also manage    */
                              /* A call to Func1 will then    */
                              /* set the prog and data bank   */
                              /* registers                    */

/* You can also define: #pragma far(Func1,1) ( where 1 is the data
  bank number)      */

EXTERN_IN_BANK(bank0_ variable);
                              /* Use this definition to make */
EXTERN_IN_BANK(bank1_ variable);
                              /* GNU recognize the p$name    */
                              /* symbol as the data bank     */
```

```
                              /* BANK_OF(name) will then give*/
                              /* the data bank number        */

                              /* Function declarations       */
void Func1_local();
typ_struct Func1();

 /*************************************** ***************/
/*                                                      */
/*Func1, a test function to show the use of prog and data banks*/
/*                                                      */
/*Comments: Return a structure to demonstrate that the stack is */
/*             preserved properly                       */
 /*************************************** ***************/
typ_struct Func1() {

 typ_struct var_func1;

 bank1_variable = 0x12;/* Do anything you need with the */
                              /* variables defined in bank 1   */

                              /* If you want do use variables  */
                              /* in another bank, first :      */
                              /* Change the data bank to the   */
                              /* right one */

 Blink_Leds_BS_DSR();         /* Just blinks leds on port2 I/O */

 Set_R_DATA_Low_Nibble(BANK_OF(bank0_varia ble));
                              /* This will set the data bank to */
                              /* one where bank0_variable is    */
                              /* located                        */

 bank0_variable = 0xAA;       /* Perform the operation you want*/
                              /* in this particular bank       */

 Set_R_DATA_Low_Nibble(BANK_OF(bank1_varia ble));
                              /* Restore the data bank register */
                              /* to the actual bank             */

 bank1_variable = 0x56;       /* Do anything you need with the */
                              /* variables defined in bank 1   */

 Func1_local();               /* Use local function declared   */
                              /* in the same bank if you need  */

 Blink_Leds_BS_DSR();         /* Blink leds                    */
 Set_R_DATA_High_Nibble(0x55);
                              /* Set High Nibble of BS_DSR to 5*/
```

```
                                /* this allows you to test that  */
                                /* return from function, the     */
                                /* value on port 2 is kept       */

  Set_R_PROG_High_Nibble(0x66);
                                /* Same thing with BS_PSR        */

  var_func1.var_a = bank1_variable;
                                /* Sets the return value       */
  var_func1.var_b = 15;         /* to any value to show the    */
                                /* stack is properly returned  */

  return(var_func1);

};/* end Func1 */

 /*************************************** ***************/
/*                                                       */
/* Define any local function you like and use it normally       */
/*                                                       */
 /*************************************** ***************/
void Func1_local(void){/* Do not declare this function  */
                                /* as far, because it is used    */
                                /* only in bank1                 */
  bank1_variable = 0;
}
```

7.4.5 The init.asm file

```
; This file sets the port configuration for bankswitch
; and the memory test board

 .include ”..\include.st9\st905x.inc”
 .text
;
;# init P0,P1,P6,P2.0,P5.3 for bank switching : c2=0,c1=1,c0=1
;
;******************* ********************************** *******;
;****  FUNCTION: init_bank_hardware                      *******;
;******************* ********************************** *******;

init_bank_hardware::

;#P0C0,1,2 = R240,241,242 Page 2
;
; Alternate function, push-pull, TTL
;
 spp #P0C_PG; set port 0 page
 ld P0C0R,#0xff
 ld P0C1R,#0xff
 ld P0C2R,#0x00

;#P1C0,1,2 = R244,245,246 Page 2
;
; Alternate function, push-pull, TTL
;

 spp #P1C_PG; set port 1 page
 ld P1C0R,#0xff
 ld P1C1R,#0xff
 ld P1C2R,#0x00

;
;#P5C0,1,2 = R244,245,246 Page 3
;
; Alternate function, push-pull, TTL, only for bit 3
;
 spp #P5C_PG; set port 5 page
 or P5C0R,#8
 or P5C1R,#8
 and P5C2R,#(~8)

;#P6C0,1,2 = R248,249,250 Page 3
;
; Alternate function, push-pull, TTL
;
 spp #P6C_PG; set port 6 page
```

```
 ld P6C0R,#0xff
 ld P6C1R,#0xff
 ld P6C2R,#0x00

;#P2C0,1,2 = R249,250,251 Page 2
;
; Alternate function, push-pull, TTL
;
 spp #P2C_PG       ; set port 0 page
 ld P2C0R,#0x00    ; HIGH NIBBLE is set to
 ld P2C1R,#0xf0    ; Output/PP/TTL
 ld P2C2R,#0x00    ; Low nibble is set to
                   ; BID/WP/TTL


; Set P/D on P5.3 on memory test board
;                  <=>
; Set segment 08 and 09 in program space

 spm               ; set program memory space
 ld  BS_PSR,#08h   ; load bank register with 08 to select 8th bank
 ld  8000,#06Fh    ; load 6F in this bank (just a write only register)
 ld  BS_DSR,#09h   ; load bank register with 09 to select 9th bank
 ld  8000,#0EBh    ; load EB in this bank (just a write only register)
 clr BS_PSR        ; reset the program space bankswitch register
 clr BS_DSR        ; reset the data space bankswitch register
 sdm               ; reset the data memory space

 jx end_init; end init_bank hardware
```

7.4.6 The mystatic.file

```
;******************** ********************************** ********
;
; This file contains the stub declaration for the nibble mode
;
; Note: _static_.s is include at the end of the file
;
;******************** ********************************** ********

__ret_far::
  pop   r4          ;saved value of R227
  and   r4, #0x0F   ; keep only bank nibble
  and   R227, #0xF0 ; raz bank nibble without modifying port nibble
  or    R227, r4    ; set bank nibble without modifying port nibble
  ret

  .macro __jp_far funcname
funcname::
  push R227         ; save value of R227
  and  R227, #0xF0  ; keep only port nibble of R227
  or   R227,#p$f$'funcname
                    ; set bank nibble without modifying port nibble
  jx   f$'funcname  ; jump in function in dynamic bank
  .endm

__ret_xfar::
  popw rr4          ; saved value of RR226
  andw rr4, #0x0F0F ; keep only bank nibble
  andw RR226, #0xF0F0
                    ; raz bank nibble without modifying port nibble
  orw  RR226, rr4   ; set bank nibble without modifying port nibble
  ret

  .macro __jp_xfar funcname,dataname
funcname::
  pushw RR226
  and  R226, #0xF0  ; keep only port nibble of R226
  or   R226, #dataname
                    ; set bank nibble without modifying port nibble
  and  R227, #0xF0  ; keep only port nibble of R227
  or   R227, #p$f$'funcname
                    ; set bank nibble without modifying port nibble
  jx   f$'funcname
  .endm

  .include "_static_.s"
```

7.4.7 The hardware.gdb file

```
#******************** ******************************
#If ST9 Emulator is an HDS model then the debugger
#will automatically use the following target SDBST9:

iftarget SDBST9
  # set general ST9 parameters
  sdb set clock 24
  sdb set cpu romless
  # set general emulator parameters
  sdb set P6 address
  sdb set dm on
  sdb set wpmf off
  sdb set nemf on
  sdb set bswmode disable

  #Set the port 2 as bankswitch port in nibble mode (4 bits)
  sdb xport 4 1
  sdb xport 5 1

  # set ST90R91 mapping
  sdb map 0 7fff uwe
  sdb map 8000 ffff uwe
  sdb map /0 /7fff uwe
  sdb map /8000 /ffff uwe

  # reset
  sdb reset

  # set port 5 bit 3 as alternate function PD
  sdb sr ea 3<<2
  sdb sr f4 08
  sdb sr f5 08
  sdb sr f6 00

  # validate banks on test board
  sdb sr e3 08
  sdb sm 0x8000 0x6f
  sdb sr e3 09
  sdb sm 0x8000 0xeb

  # Set the bank 0 for default bank
  set RR226=0

endiftarget
#end iftarget SDBST9
#******************** ***************
#******************** ******************************
#If ST9 Emulator is an ST9xxx model then the debugger
```

```
#will automatically use the following target SDB9XXX:

iftarget SDB9XXX

  #Set the port 2 as bankswitch port in nibble mode (4 bits)
  bankswitch nibble

  pd_signal used
  map p:0 0x7FFF sw
  map p:1:0x8000 0xFFFF sr
  map p:0:0x8000 0xFFFF sr

  map d:0 0x7FFF sw
  map d:0:0x8000 0xFFFF sw
  map d:1:0x8000 0xFFFF sw

endiftarget
#end iftarget SDB9XXX
#******************** ***************
```

7.4.8 The page.gdb file

```
# load modules banks
source page.bl9

# set internal GDB9 bank number
set bank_prog_and 0x0F
set bank_data_and 0x0F
```

NOTES:

Software examples included in this note are intended for guidance only.
SGS-THOMSON shall not be held liable for any direct, indirect or consequential damages with respect to any claims arising from use of such software.