FIELDBUS FOUNDATION Field Device Software
PROFIBUS PA Device Software

Generation Process User Manual

Version 2.41

Date: November, 2011

SOFTING Industrial Automation GmbH
Richard-Reitzner-Allee 6
D-85540 Haar
Phone (++49) 89 45656-0
Fax (++49) 89 45656-399

Table of Contents

Terms and Abbreviations

CPU	Central Processing Unit
EEPROM	Electrically Erasable Read Only Memory
FBIF	Function Block InterFace
FBL	Function Block Layer
FD	Field Device
FDC	FF Communication Layer
FF	Foundation Fieldbus
ID	Identifier
OS	Operating System
OSIF	(Softing FD) Operating System Interface
PA	(PROFIBUS) Profile for Process Automation
PAC	PROFIBUS-PA Communication Layer
RTOS	Real Time Operating System
MVC	Multi Variable Container
NV	Non Volatile

References

/1/ GNU Make Manual Version 3.79, 04 April 2000
(e.g. http://www.gnu.org/software/make/manual/make.html)

1 File Structure

1.1 Directory Structure

The directory structure is basically similar in each FD software project, independent of the certain protocol. Furthermore, for the Softing main development tree both protocols are contained in one common file structure.

Directory name	Description
`_dev_tools`	Contains all GNU tools needed for software generation
`_doc`	FD related documentation
`base`	Softing main field device software branch; the central parts delivered are in object code (for object code customers). The function blocks and the SYS components are always delivered in source code.
`appl`	Application specific unit (as seen from the protocol software point of view); delivered always in source code.
`ffbl`	Function block components belonging to the FF protocol
`pfbl`	Function block components belonging to the PA protocol
`fdc`	FF Device Communication layer unit
`cif`	Communication layer Interface component contains the initialisation and main entry routines of the FF protocol layer
`dll`	Data Link Layer component
`fal`	Fieldbus Application Layer component
`fbs`	Function Block Shell component of the FF protocol layer
`fmg`	Fieldbus Management component
`uta`	Upper Tester Agent component
`inc`	FDC internal component spanning header files
`mak`	FDC related makefiles
`pac`	PA Communication layer
`cif`	Communication layer Interface component contains the initialisation and main entry routines of the PA protocol layer
`dps`	DP Slave component
`dusr`	Example of a DP Slave application
`fbs`	Function Block Shell component of the PA protocol layer
`fdl`	Fieldbus Data link Layer component
`inc`	PAC internal component spanning header files
`mak`	PAC related makefiles
`sys`	Unit containing system level components
`dbg`	Debug output component
`eep`	EEPROM Interface component
`osif.xxx`	Operating System Interface component, specific to OS xxx. `osif.emb` as default for embOS by the company Segger.
`inc`	Global header files of the protocol software
`ff`	FF specific global header files
`pa`	PA specific global header files

Directory name	Description
mak	Global fixed makefiles; included by the other makefiles
(CPU)	CPU type dependent makefiles; included by the other makefiles
target<_n>	All files that are dependent on the target hardware or target application, i.e. which have to be adopted within the project implementation. In general, all files of the target branch are delivered in source code, except project specific branches of files that normally belong to base/fdc and base/pac
appl	Target application specific parts
fbif	FF function block interface component including device specific DD files and GenVFD scripts
fdev	Device initialisation, startup and transducer blocks examples.
pbif	PA function block interface component including device specific GenVFD scripts
pdev	Device initialisation, startup and transducer blocks examples
cfg	Global Device configuration
hw	(Target) HW specific component
sys	Target specific files of the SYS components
eep	EEPROM Interface component
osif.xxx	Operating System Interface component, specific to OS xxx
inc	Global target specific header files
mak	Central project generation directory containing the main makefile; which may be started directly
(var_1)	Generation environment for variant “var_1”. The makefile may be started directly or via of the central makefile.
<(var_n)>	Generation environment for variant “var_n”. The makefile may be started directly or via of the central makefile.

One Project may contain several (i.e. more than one) target branches. These branches may differ in the target hardware or application, but share one and the same version of the FD protocol software. If several targets refer to different versions of the protocol software, it is advised to use different projects with a separate base branch instead of one project with several targets.

All components, except those in target/appl, share the same subdirectory structure:

Directory name	Description
(comp)	
inc	Component specific internal header files
src	Source code files of the component
mak	Component specific makefile. Cannot be started directly, but will be called by the central makefiles.
appl/fbif	
inc	Component specific internal header files
src	Source code files of the component
ddl	FF specific Device Description files

Directory name	Description
script	Device specific GenVFD scripts for FF
mak	Component specific makefile. Cannot be started directly, but will be called by the central makefiles.
appl/pbif	
inc	Component specific internal header files
src	Source code files of the component
mak	Component specific makefile. Cannot be started directly, but will be called by the central makefiles.
script	Device specific GenVFD scripts for PA
gsd	PA specific Device Description files

1.2 Makefile Structure

Directory / File Name	Description
base	
mak	
makefile.mak	Entry point to generate the FDC / PAC units
makeenv.mak	Entry point to include the processor specific makefiles
(CPU)	CPU = M16C
makerules.mak	Global compiler specific make rules
makedefaults.mak	Global compiler specific command line options for the several development tools
(unit)	unit = FDC, PAC
mak	
makefile.mak	Unit specific makefile; generates all components of the unit, maybe separated by different targets
makedeps.mak	Goals and dependencies common to all component specific makefiles
(comp)	comp = dll, fal, ...
mak	
makefile.mak	Component specific makefile; contains the list of sourcefilesthat make up the component.
makefile.dps	Dependency definitions for the source files; generated automatically by GCC
target	
appl	
[f,p]bif; [f,p]dev	Application specific things
mak	
makefile.mak	Makefile which generates the [F,P]BIF component. Includes the according file makefblk.mak from base/appl/[f,p]fbl/mak, which contains the source file assignments for the function blocks
makefile.dps	Dependency definitions for the source files; generated automatically by GCC
(comp)	(comp) = hw, ...
mak	See above base/(unit)/(comp)/mak
mak	Base directory to generate the device firmware
makefile	Central entry point to generate several variants
std_makefile.mak	Common makefile to generate the FD Software. Valid for all protocols and variants
std_settings.mak	Common file defining the settings for the development tools when generating the FD Software. Valid for all protocols and variants
(var_n)	var_n = emu, release, PD30, ...
makefile	Entry point to generate the according variant

Directory / File Name	Description
`makesettings.mak`	Project / target / variant specific command line options for several of the development tools
`makeconfig.mak`	Variant specific target configuration options

2 Makefile Basics

As MAKE tool, the program GNU make in conjunction with some additional utilities out of the 'CYGWIN unix tools for WINDOWS' suite is used. This implies some special rules and properties, not very common to the "standard" WINDOWS user:

- File names distinguish small and capital letters
- Subdirectory names are separated by / (not by \ !)

The directory where the CYGWIN tools are located must be included in the computer's PATH environment variable.

Additionally, some environment variables are used to pass the location of the compiler and the operating system to MAKE. Currently there are used:

- CC_M16C Path to M16C compiler
- EMBOS Path to the Segger embOS (for M16C)

Note: The path must not contain any blanks.

To avoid any confusion between the application targets as mentioned in the previous chapter and the "targets" of a makefile as explained in the make documentation, all makefile targets are referred to as "goals" in the subsequent text.

The make environment of the Softing FD software respects the following conventions:

- Directly executable makefiles have the name `makefile`
- All other makefiles included by `makefile` have the file extension `.mak` (except the automatically generated dependency include files, which are named `makefile.dps`). They must not be executed directly, even if a single component is to be generated.
- All makefiles contain facilities to generate the software (goals `all`, `(comp)`, ...), to cleanup the directory structure from generation results (goals `clean`, `clean_(comp)`, ...) and to deliver the source files according to the project settings (goals `deliver`, `deliver_source`, `deliver_objects`, `dlv_(comp)`, ...).
- Except for development tools (which are in general addressed via environment variables), all directory definitions are relative to the directory where the variant generation makefile resides. This implies, that all variants should be located on the same level as subdirectories of `target/mak`.
- All project / target / variant specific settings are contained in the files `makefile`, `makesettings.mak` (concerning the compiler, linker ... options) and `makeconfig.mak` (containing the most frequently used configuration settings), which are located in subdirectory `target/mak/(var)`. Settings common to all Softing FD projects are located in `base/mak/(CPU)/makedefaults.mak`.

According to the syntax of the program GNU make all goals that are not files created during the generation process are defined as `.PHONY` at the beginning of the makefiles to stop make from searching these (never existing) files.

All makefiles define their search path to find source files in such a way that files in the target branch are used primary to the ones in the base path. This allows the makefiles to locate project specific adoptions to files of the mainstream development in a subdirectory on the target side with the same name as that on the base side. No adoptions to makefiles have to be performed to include the changed source file, nor do any files in the base branch have to be changed. All changes are located exclusively on the target side.

This fact applies to the makefiles itself as well. The call of subsequent makefiles is written in such a way, that the target branch is evaluated for the appropriate makefile first and then the base branch is evaluated. Thus, if any of the base components has to be extended by a certain file, the appropriate makefile should be transferred to the target branch and adopted here.

Example: If (for any reason) the algorithm of a PID function block (which is located in file `fblk_krn.c` of the FFBL component) has to be changed, the adopted file `fblk_krn.c` should be located in (the newly created) directory `target/appl/ffbl/src`. This file will be automatically used instead of that one in `base/appl/ffbl/src`.

Example: If the DLL component has to be extended by a project specific adoption, the according source file shall be located in `target/fdc/dll/src`, then the makefile, which is responsible for generating the DLL component, shall be copied to `target/fdc/dll/mak`, before any changes are introduced to insert the new source file into the make process.

3 Makefile Descriptions

3.1 Central Entry Point `target/mak/makefile`

This file is only intended to provide a single entry point for the generation of all variants. Therefore it always contains, beneath some make specific stuff, the global goals `all`, `clean` and `deliver`, which simply call all variant makefiles with the according goals. To simplify handling with certain variants, additional goals may of course be defined (e.g. `emu`, `release`, `uta_emu` ...).

All valid goals shall be mentioned in an error message, which will be displayed if called with an invalid goal. If called from another (higher level) makefile, some additional information is displayed concerning the currently processed target application.

3.2 Generating Variants

The file `target/mak/(var_n)/makefile` is responsible for generating, but also cleaning and delivering the complete software for the variant `var_n` of the project. It contains the directory definitions as well as the definitions to generate the several subcomponents. To do its job it uses the supplemental files `makesettings.mak` (containing the variant specific settings for the development environment, e.g. compiler, linker settings etc.), `makeconfig.mak` (providing the most common used target configuration settings) and the linker definition file, where all of these files have to be located in the same directory.

3.2.1 Variant specific Makefile

In most cases the `makefile` will not differ from variant to variant. All important variant specific settings are sourced out to the files `makesettings.mak` and `makeconfig.mak`, to simplify handling for the user.

Beneath some global make control options at the very beginning of the file the variable *`USED_MAKEFILES`* is defined and via the export directive, passed to all subsequent makefiles. This is used to automatically set up a list of all makefiles, which are involved in the generation process of a certain component. The component will be rebuilt, if any of the according makefiles has been changed.

Now some directories are defined, that the subsequent makefiles and procedures rely on. In general there is no need to change them, with one exception:

`EXE_DIR` specifies the directory where the software executable will be created. Normally this is the subdirectory exe. For some emulators it is necessary to change this, so as to enable them to find and associate the source files to the code. E.g. the PD30 emulator for the M16C requires the resulting .x30 file to be located in the current directory, i.e. from where the source compilation has been started. All object files are written to directory *`OBJ_DIR`*, while the libraries generated from the objects as well as those delivered with object code licenses will be found in *`LIB_DIR`*. If any listing files are generated during compilation, they are written to *`LST_DIR`*.

Next in this section the names of the executable and the according linker definition file are compiled from the several settings made before.

Finally, some definitions are made to compile the delivery out of the project tree. This feature will be used only by Softing.

The subsequent section defines the main dependencies. First of all, any initialisation issues have to be solved here (as make executes the according commands in the sequence it finds in the dependencies).

As the goal `all` is the default target of all makefiles described here, it must be located as the first goal within the dependency list. This is done by including file `Makeenv.mak`. Therefore it is imperative to insert goal before the `include` statement, otherwise the execution of `make`, without any selected options, will not generate `all`!

Now all main goals of the makefile are defined one after another.

As last section, the error message is defined, which will be displayed, if the makefile has been called with any unknown goal. Certainly the message should comply with the valid goals, which have to be named here explicitly.

3.2.2 The File `makeconfig.mak`

As explained above, first of all the current file name is added to *`USED_MAKEFILES`* in order to guarantee a correct generation process.

Next, *`TARGET_NAME`* is defined, which must correspond to the variant name `var_n`. It is used for clean and deliver operations.

Now some configuration variables are defined, which are used throughout the whole generation process to select the appropriate source files. Some development tool options, as defined in `makesettings.mak`, are dependent on these configuration switches. As make is case sensitive, the correct spelling in upper / lower case is important!

Table 3-1

Variable	Values	Description
`ACTIVE_MODE`	`DEBUG` `RELEASE`	

`HW_TYPE`	`BFD_HW`
	`FBK_HW`
	`FBK2_HW`
		Defines the hardware platform

`CPU`	`M16C`	Defines the CPU type of the target. It is used to find the appropriate rules and defaults
`CFG_SYS_DIV`	`2` `4 (only for BFD_HW)`	Defines the CPU clock prescaler value
`CFG_ISR_SYS_DIV`	`2` `4 (only for BFD_HW)`	Defines the CPU clock prescaler value for FBC interrupt routine
`FB_CHIP`	`FIND1P_DMA` `SPC42`	Defines the fieldbus controller chip. Used to select the right sources in the data link layer components. Use FIND1P_DMA, if the UFC100 fieldbus controller in DMA mode is used.
`OSIF`	`osif.emb`	OSIF component to be used. Must of course comply with the operating system library
`[...]`		Optional additional settings depending on the OS

Variable	Values	Description
FIELD_ DIAGNOSTICS	NO_FIELD_ DIAGNOSTICS USE_FIELD_ DIAGNOSTICS	Enables Field Diagnostics functions. For devices tested with ITK 6 Field Diagnostics is mandatory
MVC_TYPE	NO_MVC USE_MVC	Defines the usage of MVC. FF only.
SW_DOWNLOAD_TYPE	NO_SW_ DOWNLOAD USE_SW_ DOWNLOAD SW_DOWNLOAD _DEV	NO_SW_DOWNLOAD is the standard setting. USE_SW_DOWNLOAD enables software download functions. SW_DOWNLOAD_DEV generates firmware with reduced functionality.
LICENSE_TYPE	USE_SEC_ EEPROM NO_SEC_ EEPROM	Customers who licensed object code have to integrate a PROM on their HW-platforms. The PROM contains a licence key; without this key the field device software is not fully functioning. Softing's FBK and FBK-2 platforms have got the PROM by default.
ADD_COMM_COMP	HM MM	Enables additional HART or Modbus master functions. HART or Modbus master functions requires addition module that are not part of the standard field device software.
NV_DATA_STORAGE	NV_DATA_DOUB LE_BUFFER NV_DATA_SING LE_BUFFER	Used to enable the NV data double buffer storage in non-volatile memory. If it is not used.
STACK_TYPE	PA FF_BFD FF_LM	Communication stack type: PA device FF basic device FF link master device FF link master functions requires addition module that are not part of the standard field device software.
BLOCKS	FF: ai, ao, di, do, pid, m_a_i, m_a_o, m_d_i, m_d_o, ar, it, sc, is PA: ai, ao, di, do, tot	Defines the block types which will be part of the function block application. The block types must comply with the blocks included in the GenVFD script.
MAIN_SCRIPT	<name>.gw	Main script for the generation tools GenVFD_FF/PA. This file contains information about data structures, VFD and block definitions.

Variable	Values	Description
DEVICE_TYPE	XXXX	FF: Defines the device type. Must comply with the device type defined in the device description. PA: Defines a macro that is applied in target\appl\pdev.
DEVICE_REV	XX	Defines the device revision. Must comply with the device revision defined in the device description (only for FF devices)
TBLK_SCRIPTS	ai_tb di_tb ao_tb do_tb mai_tb mao_tb mdi_tb mdo_tb ...	Defines the transducer block types which will be part of the function block application. May be more than one. Caution! The block types must comply with the blocks included in the GenVFD script! There can be added also manufacturer specific transducer blocks.
FBL_TYPE	FBL_STD FBL_BLOCK_ INST	Defines if the FF function block layer supports - block Instantiation: FBL_BLOCK_INST - no block instantiation, capability levels as an option: FBL_STD Block instantiation requires addition module that are not part of the standard field device software.
IAR_COMPILER_VER	136 32x 35x	Defines the version of the IAR C Compiler. Softing's standard compiler for FF/PA stack is version 3.50 (35x).
MEM_MODEL	m, f, h for compiler version 136 or far, huge for compiler version 3xx	Memory model of the compiler to be used. Depends certainly on the target CPU type, as it will be passed directly to the compiler command line. For M16C : (m)ixed, (f)ar, (h)uge (for compiler version 136) or far, huge (for compiler version 3.xx).
DELIVERY_TYPE	OBJECT SOURCE FBK	For Softing internal use only to compile project deliveries
LINK_FILE	<file>.xcl	
BFD_EXECUTABLE *UTA_EXECUTABLE*	<file>.mot	Defines the Names of (executable) firmware files.

Variable	Values	Description
[...]		Lists of software components

As follows, some filenames are defined, setting the main output files of the generation process and the according linker definition file. The output files will reside in the subdirectory as specified by the variable *EXE_DIR* in `makefile`.

Finally, all components to be included into the generation process have to be listed here. The `makefile` relies on the variables *nn_COMPS* (with *nn* = *PA, BFD, UTA*) and (for the matter of cleaning) *ALL_COMPS*. These variables must be set here.

3.2.3 The File `makesettings.mak`

As already mentioned, `makesettings.mak` contains all project rsp. variant dependent settings for the development environment. Examples are optimisation settings for the compiler, symbol information inclusion for the linker etc.. Of course, these settings mainly depend on CPU and compiler type. In general, the variables *CCFLAGS*, *ASSEMBLERFLAGS*, *LIBRARY_MANAGERFLAGS* and *LINKERFLAGS* may be extended here (Note the '+=' assignment! Otherwise the previous content will be lost!); their basic content is defined in the CPU specific file `base/mak/<CPU>/makedefaults.mak`.

The command line 'define' statements for the compiler / assembler (in most cases selected with a –D command line option) must comply with the configuration settings as made in the file `makeconfig.mak`. E.g. if SPC4-2 is selected as the fieldbus controller chip, the according command line switch '*FC_SPC42*' must be added to *CCFLAGS*.

3.2.4 Linker Definition File

As file `makesetting.mak`, also the linker definition file is highly depending on the development environment. Mainly, memory areas are defined here and assigned to the code and data segments, as well as some global processing options of the linker.

The protocol software assigns (as far as supported by the according compiler) each component to its own code segment and certain data segments, thus allowing an easy overview on the memory requirements of the components. The linker definition file now has to map the several code and data segments to the available memory areas. The code segments are usually named '[COMP]_code', where [COMP] is replaced by the certain component names ('fdl', 'dps', etc).

3.3 Global Settings Concerning to the Development Environment

As the global settings for the development environment are common to all FD projects (as long as they are not very specific to one customer adoption), they are related to the base branch. For each supported development platform, there exists a subdirectory with name of the CPU type in `base/mak`, which contains the two files `makedefaults.mak` and `makerules.mak`. The makefiles which perform the generation just have to include `base/mak/makeenv.mak`, which on its part includes the appropriate (according to the *CPU* setting made in `makeconfig.mak`) files as mentioned previously.

The file `base/mak/makefile.mak` contains the central entry point to generate the central protocol software libraries. Depending on the protocol type it branches to the FDC or PAC part of the case components.

3.3.1 The File `makedefaults.mak`

This file defines the global (i.e. common to all projects) command line switches for all programs of the development tool chain as well as the directories, where these tools are found. Additionally, a dependency to the goal 'all' is settled, to make sure, that the goal 'all' will be generated by any makefiles, if no certain goals are defined on the command line. As previously mentioned, for this reason `makedefaults.mak` must be included (via `makeenv.mak`) in all makefiles before any goal is defined.

3.3.2 The File `makerules.mak`

The file `makerules.mak` contains implicit rules to generate object files from source files, to combine object files into libraries and to link the libraries to an executable. Finally, it contains a rule to automatically generate a list of prerequisites for each C source file (using GCC). These prerequisites (in fact, the tree of files included by the C source) are written to a file named `makefile.dps`, which exists for each component in the `mak` subdirectory (see below).

3.4 How the Components are generated

The component specific makefiles (named `makefile.mak`) finally compile the source files to objects and combine these objects to a library file, which has the name of the component and the library specific extension as defined in `makedefaults.mak`. This library file will be created in the directory as specified in the root `makefile` by the variable *`LIB_DIR`*.

3.4.1 Similarities to all Components of a Unit

As entry point to a unit the file `base/[unit]/mak/makefile.mak` contains (similar to the variant root `makefile`) the following information: what components the unit consists of, how they have to be generated and what libraries will be created.

For the basic protocol software components themself, the makefiles are very similar. In fact, they only differ in the list of source files making up the component. All common parts are combined into the file `base/[unit]/mak/makedeps.mak`. In addition to some directory settings it contains all dependencies to be regarded when generating a component.

3.4.2 Component specific Makefiles

As mentioned above, the makefiles for the protocol components of the FD software are very simple. They just contain the list of source files that belong to the component. For delivery reasons, additionally the list of header files relative to the component may be present.

The makefiles for the supplemental components, e.g. EEP or OSIF, are slightly more complex, as they do not refer to any common file for the directory settings and dependencies, but directly contain all the necessary settings. They are in most cases located in the target branch of the project tree, but may reside on the base branch also. Nevertheless, they work in the same manner as those for the protocol components, with the exception of the one for the 'FBIF' component, which will be explained in the next chapter,.

3.4.3 Generating the FBIF/PBIF Component

The generation of the FF 'FBIF' and PA 'PBIF' differs in some details from all other components. The sources are divided into the base part containing the kernel of the function blocks with the interface to the protocol software and the actual FBIF or PBIF part, where the GenVFD tool comes into usage. The base sources are located in the branch `base/appl/[f,p]fbl`, while the FBIF part, which is completely project dependent, is found in `target/appl/[f,p]bif`. The according makefile in `target/appl/[f,p]bif/mak` has to run the GenVFD Tool to generate the according C source files from the block descriptions, then compile these sources as well as the kernel sources and combine them into the resulting FBIF library.

The supplemental file `base/appl/[f,p]fbl/mak/make[f,p]fbl.mak` contains the assignments, which sources concern to the certain function blocks. As already described, the list of function blocks is set up in the file `makeconfig.mak` and is evaluated here to transform it into the list of source files as well as their according header files and the related block script . This list is needed later on during the compilation process.

To ensure the version compatibility between the GenVFD tool and the according script files, the tool should be located in the same directory `target/appl/[f,p]bif/script`. It takes the scripts together with the symbol file as generated during the tokenizer process of the device description and compiles them to C code files in the `src` directory of the component. After running GenVFD, the recursive call of make re-estimates all existing files and their dates, now having the complete list of all files available, as needed. The C sources are compiled and the library is generated.

3.4.4 Device specific component(s)

The FD software structure allows encapsulating the protocol and the function block parts almost completely from the device application part. The main interface is within the transducer block(s). The transducer functionality as part of the Softing deliverable may be seen as sample implementation to show the basic usage and also to demonstrate the generation process. This sample transducer is called the [F,P]DEV component.

Generating the device specific software and integrating it with the protocol components is shown along with the sample DEV component. If needed, the device application software may certainly be separated into several components; the generating process basically stays almost unchanged.

The file `target/appl/[f,p]dev/mak/makefile.mak` looks as follows:

```
.PHONY: $(COMPONENT) clean_$(COMPONENT) dlv_$(COMPONENT)
USED_MAKEFILES += $(TARGET)/appl/$(COMPONENT)/mak/Makefile.mak
```

Some settings to control the make process

```
###############################################################################
# defaults
COMPONENT = fdev
```

`pdev` for PROFIBUS-PA. This name has to be included in `makeconfig.mak` in the list of components, which are part of the FD software. It must comply with the subdirectory name and with the instructions in the variant specific main `makefile` to generate the several components (section `components`).

```
include $(BASE)/mak/Makeenv.mak
```

Set development environment: Settings and rules according to CPU type etc.

```
SRC_DIRS =  $(TARGET)/appl/$(COMPONENT)/src
INC_DIRS := $(INC_DIRS):$(TARGET)/appl/fbif/inc:$(BASE)/appl/ffbl/inc: \
$(subst /src,/inc,$(SRC_DIRS))

vpath %.c   $(SRC_DIRS)
vpath %.h   $(INC_DIRS)
```

Set directories to search for source and include files. Directory names have to be separated by colons (:), not semicolons (;)!

```
##############################################################################
# target files of this component
SRCS = appl_if.c appl_trn.c appl_res.c
```

The variable SRCS holds the names of all source files of the component that will be compiled and combined into the component's library. CAUTION! Take care on the correct spelling of the file names, since `make` evaluates names case sensitive!

```
CFGS = fdev_cfg.h
```

`pdev_cfg.h` for Profibus PA.

```
SINCS = cs_fwinf.h da_fdev.h da_fbif.h
```

Include files for segments definition. `da_pdev.h` and `da_pbif.h` for Profibus PA.

```
OBJECTS = $(addprefix $(OBJ_DIR)/, $(addsuffix $(OBJ_SUFFIX), $(basename
$(SRCS))))
```

Evaluate object file names out of the source's names

```
LIB     = $(LIB_DIR)/$(COMPONENT)$(LIB_SUFFIX)
```

The name of the component library is identical to the component's name itself, but suffixed with the linker's standard file extension for libraries. It will be stored in the directory for libraries as specified by the main variant-specific `makefile`. In the standard sample for M16C software the final name will be `fdev.lib` rsp. `pdev.lib` and will reside in the `lib` subdirectory of the variant.

```
##############################################################################
# dependencies
$(COMPONENT): $(OBJECTS) $(LIB)
$(LIB)      : $(OBJECTS)
$(OBJECTS)  : $(USED_MAKEFILES)
```

Set up the dependency cascade: component ← component library ← object files to be combined in the library ← implicitly the source files and all makefiles taking part in the generation process of the component (← = "depends on").

```
#-----------------------------------------------------------------------
```

```
# Clean up
clean_$(COMPONENT):
      @echo cleaning $(COMPONENT)
      -@rm -f $(LIB) \
            $(OBJECTS) \
            $(TARGET)/appl/$(COMPONENT)/mak/makefile.dps
```

Goal `clean_fdev` (`clean_pdev`) deletes all files created during generation of the component. Will be used during the global “clean” process.

```
#---------------------------------------------------------------------------
# Maintain dependencies
$(TARGET)/appl/$(COMPONENT)/mak/makefile.dps: $(SRCS)
```

`makefile.dps` contains the dependencies of source files and shall be renewed whenever a source file changes.

```
#---------------------------------------------------------------------------
# Deliver Sources
-include $(TARGET)/appl/$(COMPONENT)/mak/makedlv.mak
```

Include the file `makedlv.mak` to deliver the sources for this component. The ‘-‘ sign before the `include` instruction directs `make` not to output an error message, if the file does not exist.

```
###########################################################################
# source file dependecies
-include $(TARGET)/appl/$(COMPONENT)/mak/makefile.dps
```

Use the source file dependencies.