Technical Note

GPF SOFTWARE RELEASE APPROACH

Document Number:	89_03_09_00506
Version:	0.4
Status:	Draft
Approval Authority:	
Creation Date:	2001-Sep-05
Last changed:	2015-Mar-08 by KTH
File Name:	gpf_memo_releases.doc

Important Notice

Change History

Date	Changed by	Approved by	Version	Status	Notes
2001-Sep-05	FR		0.1		1
2001-Nov-5	FR		0.2		2
2003-Jul-3	FR		0.3		3
2004-Apr-19	KTH		0.4		4

Notes:

1. Initial version
2. Change of minor number handling
3. Added gpf_updates.csi; component list update
4. Conversion to TI document format, update of component list and file locations

Table of Contents

1 Introduction **5**

2 History **5**

3 Components **5**

4 Variants **6**

5 Labels **6**

6 Directories and Files **7**

7 Combined Releases **8**

8 GPF Internal Rules **9**

8.1 Development 9

8.1.1 Branch and View Names 9
8.1.2 Working as usual 10
8.1.3 Bug Fixing 11
8.1.4 Working with Variants 11

8.2 Integration 12

8.2.1 From Development 12
8.2.2 From Bug fixes 14

Appendices **15**

A. Acronyms 15
B. Glossary 15

List of Tables and Figures

Table 1 – List of GPF Components 5
Table 2 – ASSIST Component used for release management 7

Figure 1 – Parallel new Features 10
Figure 2 – Two Developers 10
Figure 3 – Parallel Feature Development 11
Figure 4 – Bug Fixing 11
Figure 5 – Variants 12
Figure 6 – Standard Integration 14
Figure 7 – Integration of a Variant 14
Figure 8 – Bug fix Integration 14

List of References

[C_7010.801]	7010.801, References and Vocabulary, Condat AG

1 Introduction

This document describes the way GPF releases software to the other teams.
Furthermore the GPF internal development and integration process is represented.

2 History

The old rules gpf_memo_cclabels.doc and gpf_memo_cclabeling.doc are obsolete. See gpf_memo_ccproblems.doc to understand the reasons to change these rules.

3 Components

GPF releases several software components. A component is a set of files (sources and/or binaries) which is fairly "autonomous", i.e. it's binaries can be build without sources from other components (except H files) and could be used independently from the other components (except DLLs).
For each component there exists a label "class" (see 4.), a readme file and a file-list batch file (see 6.).
The current components are:

component	**content**	**home directory GPF\...**	**Resp.**
ASSIST	pseudo component for files used for release management	assist	FR
BUSYB	Build System Berlin	util\busyb	CKR
CCD	condat coder decoder	CCD	KTH
CCDDATA	CCD coding tables	CCD	HSC
CCDGEN	mdf + pdf compiler	util\CCDGEN	SIJ
FRAME	frame + all supported OS layers + TST + all supported test interface drivers	FRAME	MP
GDI	GDI documents	DOC\gdi	FR
GENTLE	Generic Test and Logging Environment	\util\gentle	SPRK
MBTN	moan button	util\moanbtn	SPRK
MSCVIEW	graphical flow viewer	util\mscview	SPRK
NTUTILS	small NT utilities (ps, pkill etc)	util\nt_utils	SPRK
PCO	viewer for traces and primitives	util\pco	SPRK
PCON	primitive converter	TST\pcon	HSC
SAPE	SAP and message editor	util\sape	TVO
SHM	Shared memory	shm_nt	RME
STACKSIZE	Stack size calculator	util\stacksize	FR
STR2IND	tool for binary tracing	util\str2ind	CKR
STRCHECK	checking generated structures	util\strcheck	LG
TAP	test application	util\tap	HSC
TAPCALLER	GUI for TAP	util\tapcaller	SPRK
TCGEN	Test case generator	util\tcgen	SPRK
TDC	Test language	util\tap\tdc	JHO
TDSGEN	test case compiler	util\TDSGEN	ES
TESTSTACK	sample protocol stack for testing the tools	util\teststack	CKR
TOOLS	3rd party tools	tools	CKR
VCMS	virtual condat multitasking system for NT	vcms-nt	RME
XGEN	word tables to mdf/pdf converter	util\xgen	ES
XM	make GUI for GPF components	util\xm	SPRK
XPAN	keyboard and display simulation	util\xpanel	SPRK
AUX	the rest	-	SIJ

Table 1 – List of GPF Components

4 Variants

Sometimes it is necessary to maintain different versions of one component in parallel (see 8.1.3.). These parallel versions of a component are called 'variants'. Normally there is only one single variant for each component, this variant is called the 'main' variant. Other variants should have speaking names, e.g. “UMTS” for a special variant of a component needed by the UMTS project.
The official version of a variant resides on a branch having the same name as the variant, in particular the main variant resides on the main branch.
Furthermore the variant name is used to indicate a variant in the label name, in the names of development and bug fix branches and in the csi-files (see 7) supporting this variant. Exception: The name of the main variant ('main') is not part of the names mentioned in last sentence.

5 Labels

We distinguish between developer labels and integrator labels. The developer labels are set by the developer of the respective component. The integrator labels are defined as result of a integration process, which is described later in this document.
The names of the **developer labels** have the following structure:
XXX_M.N.K for the main variant, otherwise **XXX_VVV_M.N.K** with

XXX	= component (e.g. FRAME),
M	= major version (updated only when the component will be redesigned),
N	= minor version (updated when new features are introduced),
K	= step number (starting with 0, updated as needed),
VVV	= name of the variant (if not 'main', e.g. UMTS)

The **integrator labels** are build from developer labels by adding the **prefix G_**.
Examples: CCD_1.6.0, FRAME_2.4.7, TAP_UMTS_1.3.2, G_CCD_1.6.0, G_TAP_UMTS_1.3.2
All these labels are never moved.

Applying such a label to a component is always done by labeling all files belonging to that component. That means, to select a specific version of a component one and only one element statement in the config spec is required.

Remark: For internal use within the GPF team some more label types are allowed:

- floating labels: The names of such contain at least the string 'FLOAT'.
  Floating labels are the only labels allowed to be moved !
  It is not allowed to use these labels for tests outside GPF !
- labels for parallel development: XXX_VVV_M.N.K_YYY_S with YYY being an abbreviation for the developed part and S being the sub version of this.
  E.g. the multiplexer driver for the FRAME has been implemented parallel to the general FRAME development (based on FRAME_2.3.6), was labeled with FRAME_2.3.6_MUX_1 and finally merged into FRAME_2.4.x.
  It should be avoided to use these labels for tests outside GPF.

All files belonging to one GPF release are labeled with a common label. To identify main releases the label has the following structure:

G_GPF_*V* ,
with *V* = current version number.

Sometimes intermediate release versions are required to update a release. GPF marks these versions with an alphabetic character following the major version indicator. These alphabetic characters increase with the amount of intermediate releases. To identify intermediate releases the label has the following structure:

GPF_*V*_*x* ,
with *V* = current version number and *x* = alphabetic character to indicate the intermediate version.

Example: G_GPF_16 (main release), GPF_16C (third intermediate release, descendant of G_GPF_16).
All these labels are never moved.

6 Directories and Files

Unless a newer directory structure is introduced the following rules apply to the GPF VOB.
Every file in the GPF VOB belongs to one and only one GPF component. All source files contain a comment near the beginning of the file indicating the component this file belongs to.
In the directory **assist** one can find all the files to understand the file/directory structure of GPF. It also provides batch support for deliveries of GPF software. In particular it contains:

file name(s)	Content
`components.html`	Component list = up-to-date version of the list above (see 0.)
`readme_xxx.txt`	readme file for component XXX, one per component, contains short description and version history for this component
`files_xxx.bat`	file lists of component XXX as batch file (see below), one per component
`labelcomp.bat`	batch for labeling all files of a component
`labelall.bat`	batch for labeling all components (used by GPF integrator only)
`gpf_mlatest.csi`	Config spec include file for selection of GPF /main/LATEST files
`gpf.csi`	Config spec include file
`gpf_VVV.csi`	Config spec include file for variant VVV, 0 or more such files
`gpf_VVV_updates.csi`	Config spec include file for variant VVV, 0 or more such files
`gpf_ir_template.txt`	Template for GPF internal integration requests

Table 2 – ASSIST Component used for release management

In the following part the most important assist-files are described in detail:

1. The relevant files for each component in a dedicated version are listed and described in the corresponding readme_xxx.txt file. gpf_mlatest.csi ensures that these are always selected in version /main/LATEST.
2. The files_xxx.bat-files can be used universally, e.g. to easily deliver a GPF component to external customers. They include at least three listings: The source file list, the binary file list and a list with all directories used by xxx. The **binary file list** contains all files which are necessary to use the component. It may also contain source files (e.g. H-files). The **source file list** contains all files, which belong to the component, but are not contained in the binary file list.

   Furthermore a list with files from other components needed by the binaries of xxx and a list with files from other components expected by the sources of xxx may be included.

   All these lists can be accessed in the way described below.

Specification of the files_xxx batch:

synopsis

```
files_xxx {S|B|D|N|E} command [parameters]
```

description

`files_xxx` applies a given `command` to every file of the selected list of component `xxx` with

S … source list
B … binary list
D … directory list
N … list with files of other components needed by the binaries (optional)
E … list with files of other components expected by the sources (optional)

In particular: for every File `F` of the respective list `command F [parameters]` will be called.

examples

```
files_ccd B echo
```
prints a list of all CCD binary files to stdout
```
files_frame S "cleartool checkin -nc"
```
checks in all Frame source files
```
files_pco D "cleartool mklabel PCO_2.2.7"
```
applies the label PCO_2.2.7 to all PCO directories
```
files_tapcaller B copy  d:\delivery\bin
```
copies all binaries of TAPCaller to the directory d:\delivery\bin.
```
files_yasc S cp -PR  d:\delivery
```
copies all source files of YASC including their directory structure.

3. labelcomp.bat can be used to label all files and directories of a component. The syntax is as follows:
   ```
   labelcomp <COMPONENT> <label>
   ```
   labelall.bat is used by the GPF integrator only to set the G_-labels.
4. GPF maintains several config spec include files described in 7. The include files used by ClearCase cannot be version controlled objects itself. Instead a particular directory on a server is used to hold these files (\\Dbgs12\csi But since it is not a good idea not to track the changes of such central files, copies of these exist in the GPF VOB under version control (Actually the files on the \\Dbgs12 server are copies of this version controlled files).

7 Combined Releases

GPF provides combined releases, i.e. releases of all GPF components which work well together. This is done by means of config spec include files which contain valid label combinations.
For GPF release *V* the config spec include file is \\Dbgs12\csi\**gpf_*V*.csi.** (V is actually the version number of \gpf\assist\gpf.csi).
These files are never changed, and the labels contained in these files are also never changed.
The official releases are updated if necessary (e.g. in case of bug fixes) in the following way: New labels are assigned to the changed components. The complete set of these labels for a release V is collected in a file \\Dbgs12\csi\**gpf_*V*_updates.csi**.
The labels are fix as always but this file will be changed if new fixes are available.
To build and test customer releases a numbered .csi file should be included. That means a config-spec of a member of the ITM team should contain a line like this:
```
include \\Dbgs12\csi\gpf_14.csi
```
The updates-files contains the updates only, i.e. you need both c.si files:
```
Include \\Dbgs12\csi\gpf_14_updates.csi
include \\Dbgs12\csi\gpf_14.csi
```
The gpf_V-updates.csi is for development only, never use it for building customer releases !

It might be necessary to release also software from parallel development branches, i.e. different variants of particular components. This will be done by special named include files: gpf_LLL_V.csi, where LLL is the name of the variant and V a version number (see 8.1.3).

Furthermore a file **gpf_mlatest.csi** exists which should always be included at the beginning (right after "mkbranch xxx") of a developers config spec. It ensures that some dedicated GPF files (like readme_xxx.txt) are always selected in version /main/LATEST. The version number is appended like this: gpf_mlates_5.csi.

8 GPF Internal Rules

This chapter is mainly meant for the GPF developers and contains very specific information and rules.

8.1 Development

In the following paragraphs the general GPF development procedure is described.
At first the naming conventions are presented and afterwards solutions for a lot of typical situations are given – emphasized by graphical examples. For each of them the following applies: The columns represent branches (titles == names of them) and the development flow goes top down marked by corresponding labels (i.e. FRAME_2.3.5). In addition dotted lines stand for development and straight lines for a merge.
Of course these graphics simplify the real situation. Normally each component consists of a number of files and directories with different version trees. For a new component release all the files and directories have to be labeled. Thus it is possible and will frequently occur that one version of a file/directory carries more than one component label. (Even float labels on the main branch may exist !)

8.1.1 Branch and View Names

The general rule is: all tasks are done on different branches. To avoid confusion, view name and branch name should be the same. At least the views shall have meaningful names.
That means also: all tasks should be done in different views since e.g. the management of different branches in one view is somewhat difficult.
The **name of the integration branch** (integration destination) is always **equal to the name of the variant**. It is therefore the main branch for the main variant (which is the only variant for most components).
The **names of the development branch/view** follow the scheme: **zy_xxx_**dev with

zy	= name (abbr.) of the developer
xxx	= name of the component

Optionally and depending on the development state (e.g. many different branches) of the corresponding component the major or even the minor version number may be included -> **zy_xxx_M.N_**dev with

M	= major version
N	= minor version

As explained in 5 sometimes parallel development of a special feature of a component is possible. To mark this in the branch-name as well an extra abbreviation may be added -> **zy_xxx_M.N_yyy_**dev with

yyy	= abbreviation for the parallel developed feature

Examples: sij_ccd_dev, rk_pco_2.3_dev, rk_frame_2.3_mux_dev, mp_frame_2.4_dev

If development of other variants then main is necessary (see 8.1.3.) the variant name has to be included in the branch name, too ->**zy_xxx_VVV_M.N_yyy_**dev, with

zy	= name (abbr.) of the developer
xxx	= name of the component
VVV	= variant name
M	= major version (optional)
N	= minor version (optional)
yyy	= abbreviation for a parallel developed feature (optional)

These branch names correspond to the labels assigned to versions on these branches:
Examples: hsc_tap_UMTS_1.0_dev, rk_pco_LTS_2.3_dev

Bug fixes are done on branches distinct from the development branches.
The **names of the bug fix branches/views** must contain the version numbers: **zy_xxx_M.N_**fix with

zy	= name (abbr.) of the developer
xxx	= name of the component
M	= major version
N	= minor version

8.1.2 Working as usual

Development of a component should usually start on a general development branch like mp_frame_dev. If new features must be developed in parallel, a dedicated branch may be created like in Figure 1.

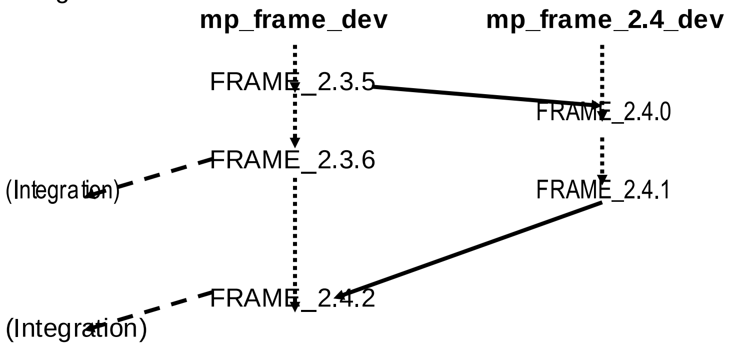

Figure 1 – Parallel new Features

Another reason for more than one development branch is obviously two or more developers working on the same component. One of them is the maintainer of the component and on his/her branch everything should be merged together at last. (see Figure 2)

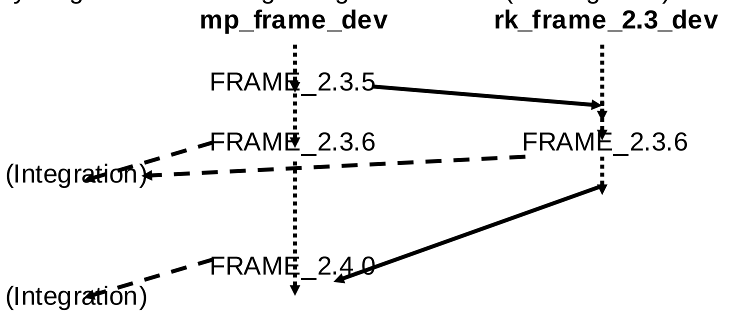

Figure 2 – Two Developers

As already mentioned in 6 sometimes a dedicated new feature will be developed parallel to the main development of a component, maybe from another co-worker. In this case label and branch names will contain the feature. Of course it's always a good idea to merge the developed stuff a.s.a.p. back to the main development. Moreover it should be avoided to announce such feature labels to other teams.

Figure 3 will give an overview of such a scenario.

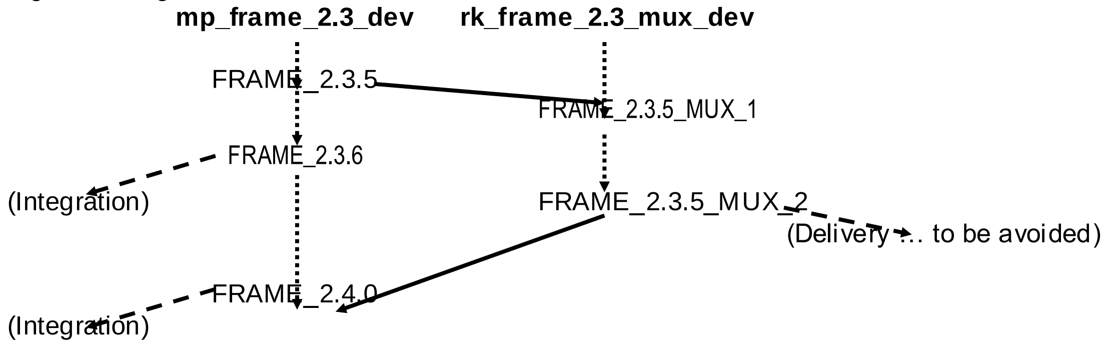

Figure 3 – Parallel Feature Development

8.1.3 Bug Fixing

Bug fixes are necessary whenever an error has been reported for a former version of a component and the newest version cannot be delivered for a reason. Thus the older version has to be fixed “in situ”.
Of course the bug fix should be merged into current development a.s.a.p.
Figure 4 shows a the way to do all this with ClearCase for a component XPAN:

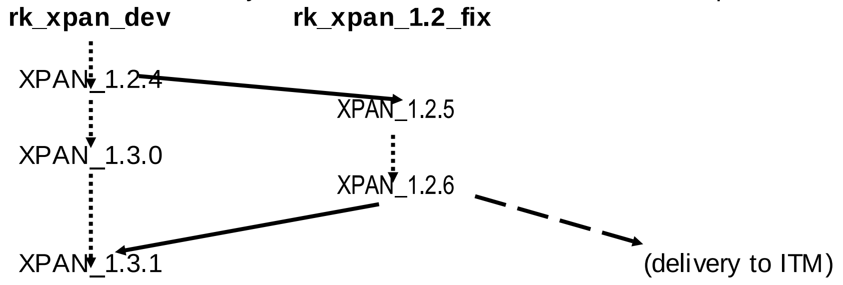

Figure 4 – Bug Fixing

See 8.2.2 for more details about the integration of bug fixes.

8.1.4 Working with Variants

Sometimes it is necessary to maintain two or more versions of a component – called variants - in parallel. What is meant here, is really development of different features in different versions, not bug fixing in older versions. Anyway, this parallel development should not occur – unless it is absolutely necessary.
Case I: A project team needs a new feature of a component. This feature is not needed by the other teams. Solution: For this component a new feature version will be implemented, containing the requested feature, but in a backward compatible way. That means, the new feature has no effect if it is not used.
In this case it is not necessary to do parallel development and therefore it is also not allowed !
Case II: A redesign of a component starts while feature development on the old version is going on. In this case there is indeed parallel development, but not on the same code. The new design gets a new major version number and development takes place as if the versions were different components (which they in fact are).
Case III: **Two teams need different new features, which cannot be developed sequentially** due to time constraints. In this case parallel development on different versions of the same code is necessary. **This case is regarded here.**
What happens is that a new branch named like the name of the variant (e.g. UMTS) is created and all component versions developed especially for this variant will be periodically integrated to this instead of the main-branch (see 12 and Figure 7).

Furthermore all development branches and labels affected by/applied to objects changed for the variant will contain the variant name, too (see 6 and 9).
Of course after some time the variant should be merged back to the main development of the component.
The concrete development procedure is illustrated for component TAP in Figure 5.

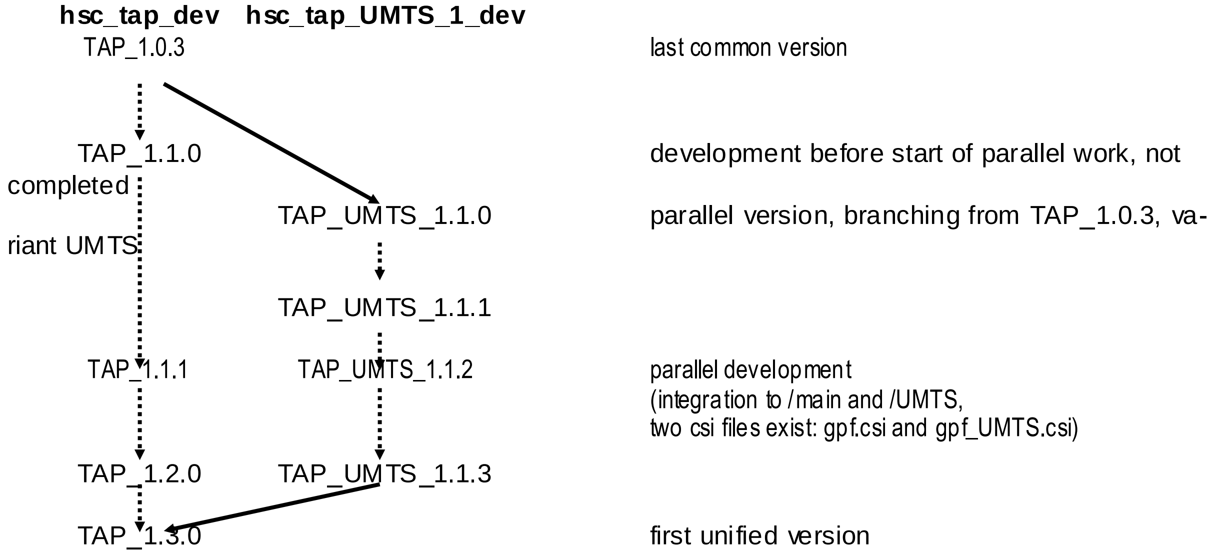
hsc_tap_dev
hsc_tap_UMTS_1_dev
TAP_1.0.3
last common version
TAP_1.1.0
development before start of parallel work, not completed
TAP_UMTS_1.1.0
parallel version, branching from TAP_1.0.3, variant UMTS
TAP_UMTS_1.1.1
TAP_1.1.1
TAP_UMTS_1.1.2
parallel development
(integration to /main and /UMTS,
two csi files exist: gpf.csi and gpf_UMTS.csi)
TAP_1.2.0
TAP_UMTS_1.1.3
TAP_1.3.0
first unified version

Figure 5 – Variants

Some more details concerning variants:
One of the parallel development branches is considered the 'main' variant. This variant is treated the ordinary way as described above.
The other parallel branches (= variants) start at a certain feature level. The labels on these branches carry the name of the variant. The minor version number starts with the increased feature level of the version, from which the branch was taken.
Except for the additional name the parallel branches (=variants) are also treated the ordinary way. The integrator maintains .csi files for all variants (see 7.).
The label of the unified version will be without the variant name, of course. This turns out to be the new main variant. Its minor version number is by one greater than the greatest in all branches.
The following **formalities** apply for such parallel developments:

- Starting a parallel development, i.e. a new variant, needs the approval of both the respective project manager and the CCL.
- Parallel variants have to be clearly marked within the integration requests (variant name).
- The history of the parallel development has to be documented within the readme file of the component (in a pseudo graphical way similar to the figure above).
- A list with all variants and their descriptions has to be updated in GPF/assist.

8.2 Integration

In this context integration stands for the GPF internal integration process which includes merging all new component files to the corresponding integration branch (usually /main), re-labeling them with G_xxx-labels (see 5) and creating a new gpf.csi-file. Afterwards this file has to be integrated by the other teams.

8.2.1 From Development

Integration normally takes place once a quarter, the current status report provided in the folder \\dbgs2\deveng\cc\gpf\status_reports notifies about the detailed project planning. Exceptions are possible in urgent cases.
If there are new features to integrate the developer assigns a fixed label to the appropriate version of her/his component and **requests for integration** to the GPF integrator.

Prerequisites:

- The readme_xxx.txt file is up-to-date (e.g. contains the description of the new features).
- The files_xxx.bat file is up-to-date.
- The new version of the component is tested and thoroughly labeled.

The **integration request** will be named like this: **zy_ir_xxx_M.N.K.txt**

zy	= name (abbr.) of the developer
xxx	= name of the component
M	= major version
N	= minor version
K	= step number

It contains the following information:
Date:
Developer:
Component:
Label:
Variant:
Bug reporter: *(customer number and ID or name, if internal)* *)
Bug report reference(s): *(Conquest/gnats/Forum Ids)* *)
Cause and short description of change:
Usable for which projects/teams:
Readme changed ? *(control question)*
xxx_files.bat changed ? *(control question)*
List of changed files:
List of changed directories:
List of binaries to rebuild:
What and how was tested ? *(Tests should be reproduceable by the integrator)*
Known dependencies:
*) in case of bug fix only (see next paragraph).

With this information the integrator merges the new releases of the components to the main branch or to the variant branch respectively and does some basic tests. At least he/she tests if everything can be compiled in all supported configurations. If everything works fine, the integrator assigns new G_xxx-labels on the main/variant branch.

The merging of all GPF components has to be done at once. This is essential due to the common directory problem (It might be a good idea to create a view containing the new versions first).

After integrating all GPF components the integrator creates a new gpf.csi file or gpf_L.csi file respectively reflecting the new release and sends a compound integration request to the integrators of the other teams.

The integrator provides the newest version of the .csi file as gpf.csi and all versions of this .csi file as gpf_V.csi, where V is the version number of the .csi file (see 7.). For variants this applies accordingly.

The GPF developers, which had released new versions, increase the minor version of their components.

This is necessary to be able to handle bug fixes. (There must be "room" for increasing step numbers.)

After all the standard integration for e.g. the component FRAME would look like this:

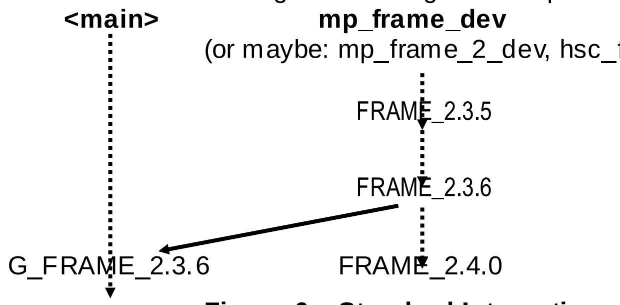

Figure 6 – Standard Integration

In case of the integration of a variant for component TAP the following picture would match:

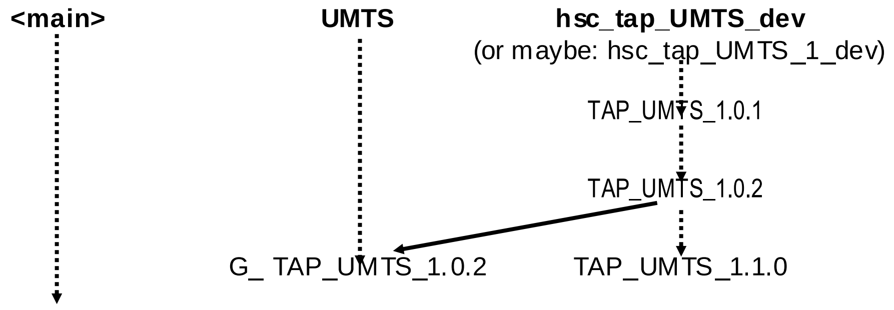

Figure 7 – Integration of a Variant

8.2.2 From Bug fixes

The integration of bug fixes is very similar to the normal process described in the previous paragraph. The differences are:

1. For urgent customer related bug fixes it might be necessary to deliver them from the bug fix branch directly, i.e. via integration request to the ITM integrator.
2. The bug fixing takes place in that version for which it was reported. All customers relying on that version will get the fixed version from the bug fix branch. The developer responsible for that component does the same fix also on the development branch (if not already done) and it will be integrated the normal way next time.
3. Two additional entries in the integration request form have to be filled out (see above).

Figure 8 pictures the integration process for a component XPAN:

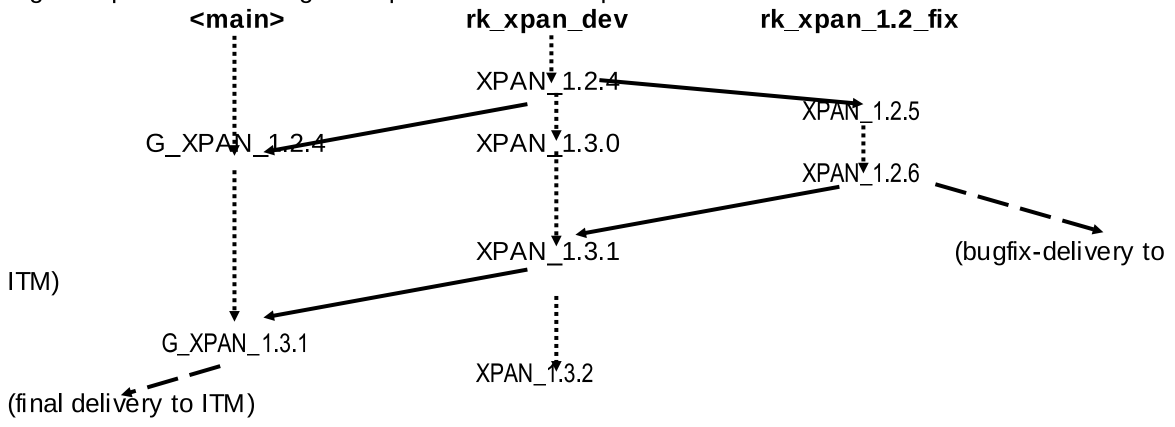

Figure 8 – Bug fix Integration

Appendices

A. Acronyms

DS-WCDMA	Direct Sequence/Spread Wideband Code Division Multiple Access

B. Glossary

International Mobile Telecommunication 2000 (IMT-2000/ITU-2000)	Formerly referred to as FPLMTS (Future Public Land-Mobile Telephone System), this is the ITU's specification/family of standards for 3G. This initiative provides a global infrastructure through both satellite and terrestrial systems, for fixed and mobile phone users. The family of standards is a framework comprising a mix/blend of systems providing global roaming. <URL: http://www.imt-2000.org/>