TARLAN_HEAT VERSION 2.0

May 2011 JM

1 Introduction

This document is about next major version of the Heating tarlan-compiler, version 2.0. The latest version in the 1.x series is 1.6. The new version brings only a few changes that are visible to the user, but internally, the compiler has been changed quite a lot. The new compiler is in a separate directory in the EROS tree, the canonical location being `/kst/eros5/dsp/tarlan/tarlan_heat_v2`. The Makefile there installs the version 2 compiler to `/kst/eros5/bin/` with the name `/kst/eros5/bin/tarlan_2.0`.

The version 2 compiler must still be considered experimental, and the EROS `tarlan` command for Heating still calls the version 1.6 compiler, `/kst/eros5/bin/tarlan_heat`, which is compiled from sources in `/kst/eros5/dsp/tarlan/tarlan_heat/`. Even though the intention is that the binary code generated by the version 2 compiler will very closely resemble the code generated by the 1.x compiler, the `tbin` and `rbin` binaries (referred together as "`xbin`" here) from the two compiler versions are quite different. That is because, in addition of the executable code that is actually loaded to RC memory, the xbin also contains a *header part*, and this has now been changed drastically. Moreover, there are also small changes in the executable code, as one of the main motivations for a new compiler version was to correct the few cases where the old compiler generates wrong code. Most especially, the TXSYNC and RXSYNC commands should now produce the expected 2 μs pulse whenever the commands are used. There are one or two other changes also, see the beginning of Section 4 in this report. Note that the version 2.0 does *not* yet implement the changes that might, or might not, be needed in connection of the Heating frequency drift problem.

These xbin changes mean that one cannot simply use some binary-comparison command like `cmp` to verify that the new compiler generates "correct" code. Moreover, the changes in the xbin header are large enough so that the current EROS `loadradar` command will not accept the binaries generated by the new compiler. I plan to update the loaders only sometime later in the summer.

To make possible testing of the 2.0 compiler against the old compiler and the present version of `loadradar`, the version 2 compiler can operate in *1x-compatibility mode*. In this mode, the version 2 compiler uses the old xbin header format, so the files can be loaded in EROS, and also the code changes are kept to a minimum (only the known bugs are fixed). At present, the compatibility mode actually is the default mode. The mode can always be turned on explicitly using the compiler option -1. Once the testing is considered complete and the loader is updated, the default mode will be changed to *native mode*. At the present, the native mode must be explicitly enabled using the compiler option -2.

To inspect the headers of the binaries, one can use standard Unix facilities like the `hexdump` program (not available in all our Suns), but the new compiler's home directory contains also a a small new program, `xbin_reader`, that can read and decode to the screen both the old and the new headers.

This document is partly a description of the new features, both internal and external, and partly a verification document for the correctness of the generated code. Section 2 explains the new features. Section 3 contains example compilations of a test program, done both with the old compiler, and with the new compiler in the compatibility mode, to verify code correctness. These runs seem to show that the 2.0 compiler is able to generate code correctly. Section 4 shows compilations done with the new compiler in native mode, to illustrate the new header format. The format is explained in detail in Section 5. Section 6 shows a compilation speed test done with a large source file.

One of the main benefits of the new header format is that it gives better support for xbins which contain multiple programs, and also, that it provides means to transmit "metadata" automatically from a tlan source file to EROS. These features appear useful enough so that all our tarlan compilers could benefit of an update to the new header format. I reckon that this can be done rather quickly, without making much change to the compiler internals otherwise. It perhaps does not make much sense to update the EROS part until all our tarlan compilers can generate the new headers.

2 What's new in 2.0

Internal features

For version `tarlan_heat_2.0`, the tarlan compiler has been internally re-organized. The compiler is still, of course, based on the highly efficient lex/yacc compiler generator that Assar started using long time ago, but the tlan file compilation is now arranged to be done in two passes per each program in the source file, instead of a single pass. Also the compiler's internal data structures have been simplified by using 64-bit variables for all AT-time related quantities and for the RC-code words themselves. For definitiveness, we now also use explicit storage size specifications like `uint64_t` when declaring any variable who's bit size does matter.

These changes result in significant simplification and clean-up of the compiler's internal logic and bookkeeping, and therefore make adding new commands both simpler and safer. Not to speak about allowing us robustly kill some old, known bugs.

However, the two-pass compilation results in marked reduction, by up to about a factor of three, of the compilation speed, and also results in larger memory consumption. A speed test done by compiling `/kst/hfexp/edmim/375Hz.tlan`, one of the largest Heating tlan files currently in use, is shown in Tables 15 and 16. With the 2.0 compiler, this worst-case compilation which produces a tbin of about 16 000 instructions long, takes about 540 ms on s2501 (Sodankylä site's ancient Sun Ultra-250 Sparc), and takes about 47 ms on a 2 GHz Intel MacBook. The older compiler version `tarlan_heat_1.6` is about three times faster, giving compilation times 170 ms and 19 ms, respectively.

The loss of speed is lamentable, and while there might exist possibilities for some performance optimization, I don't believe that the old speed can be regained. On the other hand, 0.5 s is not a very long time during the experiment preparation phase, and the new compiler is so much simpler internally that I'm inclined to accept the performance penalty.[1]

[1]But 500 ms is long enough that I'm, for the time being at least, dropping my idea of letting EROS always (re-)compiler the tlan source files at load time. This would have been a robust way to enforce tlan-xbin consistency, which currently is not checked at all.

The 2.x compilation steps are the following.

1. First, one uses lex/yacc to generate two Bit Operation Tables", one for RX, the other for TX. Both of these tables contain 4-element records with
   a) required at-time of the bit operation as a 64-bit integer
   b) operations's unique sequence number
   c) code bit number 0...63 (no separation between "high" bits and "normal instruction bits" 0...31 any more).
   d) the value to be set for that bit number at that time, either 0 or 1. [2]

   It is now very simple to handle tarlan commands that generate pulses, and it does not matter *when* that pulse should happen. One just generates two bit operations for the required times relative to the current AT-time, corresponding to the leading and trailing edge of the pulse. In the 1.x version the biggest trouble was to keep track on the overall time ordering when pulses were present. I never managed to do that correctly in `tarlan_heat_1.x` for the towards-the-future going RXSYNC and TXSYNC pulses. Now the whole ordering business is postponed to the time where it properly belongs, namely, when *all* bit operation times are available.

2. Phase (1) proceeds until lex/yacc finds an end-of-program instructions (REP). Then the Bit Op Tables are used for code-generation for that program, separately for TX and RX. The code is appended to two Code Tables, one for RX, the other for TX.
   a) The first job in code-generation is to order the Bit Op Table to increasing AT-time order based on (1.a). In case when there are several operations at the same AT-time, which is normal, one uses the sequence number (1.b) as an additional criterion to get unambiguous, natural ordering. The ordering of the records uses quicksort, and is implemented by a single invocation of the qsort() routine of the standard C-library.
   b) Once the bit operations are in their natural time order in the Bit Op Table, a simple for-loop generates the executable RC program to the Code Table, by applying the bit operations one after another to the current 64-bit RC-word. (The initial word is the default RC-word.) As soon as the loop finds that the bit-operation AT-time increases, the current word is completed by assigning the dwell-time bitfield, and then the word is appended to the Code Table. Buildup of the next word then commences. The for-loop executes until all bit-operations in the Bit Op Table are processed.
   c) After the program's executable is ready, the compiler also stores various auxiliary info like the REP, the number of instructions, duty cycles, program name, etc., to Program Info Table, to be used for the construction of the xbin header once all programs in the file have been processed.
   d) After the header info is saved, compiler counters are reset and execution moves to (1) to start compilation of the next program in the file.

[2]In compiler 1.x, one uses the bit-operation "flip the current bit value". We could do that very simply here also, but I prefer an explicit assignment.

Once the tarlan source file comes to an end, the Code Tables and the Program Info Table are used to generate the output xasc and xbin files. This may also require re-ordering of the 64-bit program code words to the big-endian byte order used by the RC.

New features visible to the user

Apart from TXSYNC and RXSYNC now compiling correctly, the visible changes from 1.x to 2.0 are mostly, but not entirely, cosmetic.

1. The 2.x compiler is source code case-insensitive, that is, the compiler does not care at all about the character case of commands. This is achieved by using the GNU `flex` tokenizer, with the flag -i, in the Makefile, instead of the older `lex` (which does not support -i).[3]

2. The 2.x compiler allows also white space as command separator when there are multiple commands in a single AT line. In 1.x, a comma is required between commands (but white space allowed in addition to the comma).

3. The 2.x compiler only requires that the AT-time does not *decrease* from AT-line to AT-line. But it can stay the same.

4. The 2.x compiler allows many, but not all, commands to have the special AT-time of zero. This means that one can in effect modify the default RC words in the tlan file. Now, as earlier, the first command at non-zero AT-time must be at AT-time 0.3 or larger, or, if the command results in a pulse, the pulse must not start earlier than 0.3.

5. The programs in the tlan file can now have symbolic names. To that purpose, there is a new command, PROG $<progname>$, that must be placed either before the very first AT-line of the source file; or between a REP-line and the next AT after an REP. That is to say, the PROG <progname> line must not be placed "in the middle of a program". $<programe>$ must start with a non-numeric character, can only contain A-Z, a-z and 0-9 and an underscore, and has max length of 32 characters.

   The given $<progname>$ is saved in the xbin header for each program in the file, and thus is made automatically available to EROS. If one does not give an explicit name, name $prog\_N$ is automatically generated for the N'th program.

6. There are some trivial changes to error messages and warnings. For instance, the compiler now complains about missing STC, BUFLIP, STFIR only if at least one channel is explicitly used (there is a CHn command) in the program.

An example illustrating the new face-lifting possibilities of source file such as the file `test1_1x.tlan` of Table 4, is shown as the file `test1.tlan` in Table 1. Please note that I definitely am not recommending using all those features in normal coding. The old, more fixed format is clean and concise, and has much to be recommended. But now the choice is on the tarlan programmer.

[3]Which of course requires that `flex` is available, which is not the case at present in most of our Solaris machines. Must be installed from `http://www.sunfreeware.com/`.

Table 1. File test1.tlan. This file requires `tarlan_heat_2.x` to compile. There are three programs in the file. The first two programs have explicit names, the last one will be given the name *prog_3* by the compiler. *Prog Long*: Commas and spaces are used to visually group together functionally similar commands. I would also argue that using lower-case letters in the commands makes numbers in commands like `stmct3&4` a little easier to read. There are now several commands at the earlier forbidden AT-time 0. And uninhibited use is made of RXSYNC and TXSYNC, which should work correctly now. *Prog Medium*: Receiver commands are in one column and transmitter commands in another column. Several commands having the same AT-time are placed to different code lines. *prog_3*: The old format is sooo charmingly from the sixties, isn't it?

```
PROG Long
%------------------------------------------
    at  0      txsync,rxsync  psavoff
    at  0.4s   rxp1on,rxp2on  stmcm1&2,stmct1&2,stmct3&4,stmct5&6
    at  0.5s   rfon*
    at  0.6s   rfoff*  rxp1off,rxp2off
    at  1.0s   psavon
  settcr 5s
    at  0      txsync,rxsync  psavoff
    at  0.4s   rxp1on,rxp2on  updm1&2,updt1&2,updt3&4,updt5&6
    at  0.5s   rfon*
    at  0.6s   rfoff*  rxp1off,rxp2off
    at  1.0s   psavon
  settcr 0
    at  10s    rep

PROG Medium
%------------------------------------------
    AT 0          rxsync       TXSYNC
    At 0.4        stfir        STMC* PROFSEL1
      at 0.4      ncosel100
      at 0.4      chl1r2
    AT 0.5        ncoprs
    AT 10         rxp2on       RXP1ON
    AT 100                     RFON*
    AT 110                     FLP*
    AT 120                     NOFLP*
    AT 600                     UPD*
    At 1000                    RFOFF*
      at 1000     rxp2off      RXP2OFF
      at 1000     rxsync       TXSYNC
    AT 2000       ch*
    AT 8000       ch*off
    AT 9998       buflip
    AT 9999       stc
    AT 10000             Rep

% An unnamed program. Will be
% named as prog_3 by the compiler.
%------------------------------------------
AT 0.4 STFIR,STMC*,RXSYNC,TXSYNC
AT 1000 REP
```

Table 2. Compiling the file test1.tlan of Table 1 with tarlan_heat_2.0, native mode. Note that native mode (option -2) is not required for the compilation itself to succeed, but here we want to produce the new, version 001, xbin headers.

```
> tarlan_heat_2.0 -vv -a -2 test1.tlan
Prog 1 (Long):       RX sa=0 len=17  bitop=43      TX sa=0 len=19  bitop=59
REP= 10 000 000.0 us    RFduty=2.0 %

AT 9999.0 STC: Channel on-times since last STC: CH1 6000.0 CH2 6000.0
Prog 2 (Medium):       RX sa=17 len=17  bitop=45      TX sa=19 len=15  bitop=67
REP= 10 000.0 us    RFduty=9.0 %

Prog 3 (prog_3):       RX sa=34 len=7  bitop=11      TX sa=34 len=7  bitop=23
REP= 1 000.0 us    RFduty=0.0 %

Parsing: 12.8 ms Codegen: 0.0 ms Filewrite: 1.0 ms Total: 15.1 ms

.rasc => test1.rasc
.tasc => test1.tasc
.rbin => test1.rbin
.tbin => test1.tbin
```

Table 3. Decoding the header of test1.tbin from the compilation of Table 2.

```
> xbin_reader test1.tbin
file     = test1.tbin
  time     = 28-May-2011 00:03:37
  version  = 1
  rc_id    = HOTtx
  nprog    = 3
  prog_header_size  = 80 bytes
  tot_header_size   = 280 bytes
  tot_code_size     = 328 bytes
prognum =  1
  name        = Long
  sa          = 0
  len         = 19
  rep         = 10000000.0 us
  rf_duty     = 2.0 %
  beam_duty   = 2.0 %
prognum =  2
  name        = Medium
  sa          = 19
  len         = 15
  rep         = 10000.0 us
  rf_duty     = 9.0 %
  beam_duty   = 9.0 %
prognum =  3
  name        = prog_3
  sa          = 34
  len         = 7
  rep         = 1000.0 us
  rf_duty     = 0.0 %
  beam_duty   = 0.0 %
```

3 Compatibility mode

The test program `test1_1x.tlan`, shown in Table 4, is a minimal modification of the file shown in Table 1. The file uses almost all commands supported by `tarlan_heat`. Only bit-operators like SBTX1 (set bit) and CBRX2 (clear bit) are not there, and neither are RXSYNC nor TXSYNC. This file can be compiled both with `tarlan_heat_1.6` and `tarlan_heat_2.0`, allowing correctness check of the compiler version 2.

Table 4. File `test1_1x.tlan`. This test file can be compared also with the version 1.x compiler. Compared to `test1.tlan` of Table 1, we can't now use AT-time 0, which makes the two 5-second sections of the the first program unnecessarily non-symmetric. Also, we need to remember that some source characters, like the "m" in `STMCm1&2`, and the "s" in some AT-times, must be in lower-case. And we can't use the `PROG` to name the programs.

```
% First program
AT 0.3      PSAVOFF
AT 0.5s     RXP1ON,RXP2ON, STMCm1&2,STMCt1&2,STMCt3&4,STMCt5&6
AT 0.6s     RFON*
AT 0.7s     RFOFF*, RXP1OFF,RXP2OFF
AT 1.0s     PSAVON
SETTCR 5s
AT  0.3     PSAVOFF
AT 0.4s     RXP1ON, RXP2ON, UPDm1&2, UPDt1&2, UPDt3&4, UPDt5&6
AT 0.5s     RFON*
AT 0.6s     RFOFF*, RXP1OFF,RXP2OFF
AT 1.0s     PSAVON
SETTCR 0
AT 10s      REP

% Second program
AT 0.5      STFIR, STMC*, PROFSEL1, NCOSEL100, CHL1R2
AT 0.7      NCOPRS
AT 10       RXP2ON, RXP1ON
AT 100      RFON*
AT 110      FLP*
AT 120      NOFLP*
AT 600      UPD*
AT 1000     RFOFF*, RXP2OFF, RXP2OFF, RXSYNC, TXSYNC
AT 2000     CH*
AT 8000     CH*OFF
AT 9998     BUFLIP
AT 9999     STC
AT 10000    REP

%Third program
AT 0.4 STFIR, STMC*, RXSYNC,TXSYNC
AT 1000 REP
```

Table 5. Compiling the file of Table 4 with `tarlan_heat_1.6`.

```
> tarlan_heat_1.6 -vv -a -o test1_1x_old test1_1x.tlan
Warning: no STFIR
Warning: no STC
Warning: no BUFLIP
Prog 1:     RX sa= 0 len= 16     TX sa= 0 len= 16   REP= 10 000 000.0 us

AT 9999.0 STC: Channel on times since last STC: CH1 6000.0 CH2 6000.0
Prog 2:     RX sa= 16 len= 17     TX sa= 16 len= 14   REP= 10 000.0 us

Prog 3:     RX sa= 33 len= 7     TX sa= 30 len= 7   REP= 1 000.0 us

Parsing: 11.4 ms total: 12.6 ms

.rasc => test1_1x_old.rasc
.tasc => test1_1x_old.tasc
.rbin => test1_1x_old.rbin
.tbin => test1_1x_old.tbin
```

Table 6. Tbin header from the compilation of Table 4. The header does not provide info about the three individual programs in the file. The 1.x compiler does not compute rf-duty cycle, leaving, the rf-duty field is zero (compare to Table 8).

```
> xbin_reader test1_1x_old.tbin
file     = test1_1x_old.tbin
  time     = ?
  version  = 0
  rc_id    = HOTtx
  nprog    = 1
  prog_header_size  = 100 bytes
  tot_header_size   = 100 bytes
  tot_code_size     = 296 bytes
prognum =   1
  name        =
  sa          = 0
  len         = 0
  rep         = 0.0 us
  rf_duty     = 0.0 %
  beam_duty   = 0.0 %
```

Table 7. Compiling the file of Table 4 with `tarlan_heat_2.0`, compatibility mode.

```
> tarlan_heat_2.0 -vv -a -o test1_1x_new test1_1x.tlan
Prog 1 (prog_1):      RX sa=0 len=16   bitop=39     TX sa=0 len=16   bitop=55
REP= 10 000 000.0 us    RFduty=2.0 %

AT 9999.0 STC: Channel on-times since last STC: CH1 6000.0 CH2 6000.0
Prog 2 (prog_2):      RX sa=16 len=17   bitop=43     TX sa=16 len=14   bitop=65
REP= 10 000.0 us    RFduty=9.0 %

Prog 3 (prog_3):      RX sa=33 len=7   bitop=11     TX sa=30 len=7   bitop=23
REP= 1 000.0 us    RFduty=0.0 %

Parsing: 11.3 ms Codegen: 0.0 ms Filewrite: 1.0 ms Total: 13.5 ms

.rasc => test1_1x_new.rasc
.tasc => test1_1x_new.tasc
.rbin => test1_1x_new.rbin
.tbin => test1_1x_new.tbin
```

Table 8. Tbin header from the compilation of Table 7. The compatibility mode uses the classic xbin header format (version=000), and can't provide for multiple programs. As a small improvement, the 2.x-compiler computes the RF-duty cycle for all programs. The classic header has place only one value, and the value shown in the header is for the first program.

```
> xbin_reader test1_1x_new.tbin
file      = test1_1x_new.tbin
  time     = ?
  version  = 0
  rc_id    = HOTtx
  nprog    = 1
  prog_header_size  = 100 bytes
  tot_header_size   = 100 bytes
  tot_code_size     = 296 bytes
prognum =   1
  name         =
  sa           = 0
  len          = 0
  rep          = 0.0 us
  rf_duty      = 2.0 %
  beam_duty    = 0.0 %
```

Table 9. Byte-by-byte comparison of tbin headers from compilations of Tables 5 and 7. The only difference between the two tbins is in the 100-byte header, where the 2.x compiler has placed the non-zero RF-duty cycle of RC prog 1.

```
> hexdump -Cv test1_1x_old.tbin
00000000  52 41 44 2d 43 4f 4e 54  00 00 01 28 48 4f 54 74  |RAD-CONT...(HOTt|
00000010  78 20 30 2e 30 00 00 00  00 00 00 00 00 00 00 00  |x 0.0...........|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000060  00 00 00 00 00 4c 4b 3e  7f f7 3f 80 00 00 00 00  |.....LK>..?.....|

> hexdump -Cv test1_1x_new.tbin
00000000  52 41 44 2d 43 4f 4e 54  00 00 01 28 48 4f 54 74  |RAD-CONT...(HOTt|
00000010  78 20 32 2e 30 00 00 00  00 00 00 00 00 00 00 00  |x 2.0...........|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000060  00 00 00 00 00 4c 4b 3e  7f f7 3f 80 00 00 00 00  |.....LK>..?.....|
```

4 Native mode (option -2)

`tarlan_heat_2.0` in native mode uses xbin header format 001, which properly supports multiple programs in the same file, and also adds the REP-time parameter, a generation-time time-stamp, symbolic program names, etc, to the header.

Also the generated code changes slightly from version 1.x. First, the 0.1μs UPD and STMC pulses now *start* at the specified AT-time, instead of *ending* at that time. Second, RXSYNC and TXSYNC work correctly, generating a 2 μs pulse starting at the specified time.

Table 10. Compiling `test1_1x.tlan` with `tarlan_heat_2.0`, native mode. Compiling with option -2 enables version 001 xbin headers, and also results in small changes in the generated code.

```
> tarlan_heat_2.0 -vv -a -2 -o test1_1x_new2 test1_1x.tlan
Prog 1 (prog_1):       RX sa=0  len=16   bitop=39      TX sa=0  len=16   bitop=55
REP= 10 000 000.0 us    RFduty=2.0 %

AT 9999.0 STC: Channel on-times since last STC: CH1 6000.0 CH2 6000.0
Prog 2 (prog_2):       RX sa=16 len=17   bitop=43      TX sa=16 len=14   bitop=65
REP= 10 000.0 us    RFduty=9.0 %

Prog 3 (prog_3):       RX sa=33 len=7   bitop=11      TX sa=30 len=7   bitop=23
REP= 1 000.0 us    RFduty=0.0 %

Parsing: 11.3 ms Codegen: 0.0 ms Filewrite: 1.0 ms Total: 13.5 ms

.rasc => test1_1x_new2.rasc
.tasc => test1_1x_new2.tasc
.rbin => test1_1x_new2.rbin
.tbin => test1_1x_new2.tbin
```

Table 11. Decoded tbin header from the compilation of Table 10.

```
> xbin_reader test1_1x_new2.tbin
 file      = test1_1x_new2.tbin
   time      = 27-May-2011 15:02:06
   version = 1
   rc_id     = HOTtx
   nprog     = 3
   prog_header_size  = 80 bytes
   tot_header_size   = 280 bytes
   tot_code_size     = 320 bytes
prognum =   1
   name        = prog_1
   sa          = 0
   len         = 17
   rep         = 10000000.0 us
   rf_duty     = 2.0 %
   beam_duty = 2.0 %
prognum =   2
   name        = prog_2
   sa          = 17
   len         = 15
   rep         = 10000.0 us
   rf_duty     = 9.0 %
   beam_duty = 9.0 %
prognum =   3
   name        = prog_3
   sa          = 32
   len         = 8
   rep         = 1000.0 us
   rf_duty     = 0.0 %
   beam_duty = 0.0 %
```

Table 12. Hexdump of the 280 byte tbin header from compilation of Table 10. The version 001 xbin header block starts with a 40-byte *master header*, after which there are 80-byte *program headers*, one for each program in the file. The header contains both ASCII strings and binary numbers, the latter being in big-endian byte order (MS byte first).

```
00000000  52 41 44 2d 43 30 30 31  48 4f 54 74 78 00 00 00  |RAD-C001HOTtx...|
00000010  00 00 00 03 00 00 00 50  00 00 01 18 00 00 01 28  |.......P.......(|
00000020  4d e0 27 ed 00 00 00 00  52 41 44 2d 43 30 30 31  |M.'.....RAD-C001|
00000030  48 4f 54 74 78 00 00 00  00 00 00 10 00 00 00 00  |HOTtx...........|
00000040  70 72 6f 67 5f 31 00 00  00 00 00 00 00 00 00 00  |prog_1..........|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000060  00 00 00 00 05 f5 e1 00  40 00 00 00 00 00 00 00  |........@.......|
00000070  40 00 00 00 00 00 00 00  52 41 44 2d 43 30 30 31  |@.......RAD-C001|
00000080  48 4f 54 74 78 00 00 00  00 00 00 0e 00 00 00 10  |HOTtx...........|
00000090  70 72 6f 67 5f 32 00 00  00 00 00 00 00 00 00 00  |prog_2..........|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000b0  00 00 00 00 00 01 86 a0  40 22 00 00 00 00 00 00  |........@"......|
000000c0  40 22 00 00 00 00 00 00  52 41 44 2d 43 30 30 31  |@"......RAD-C001|
000000d0  48 4f 54 74 78 00 00 00  00 00 00 07 00 00 00 1e  |HOTtx...........|
000000e0  70 72 6f 67 5f 33 00 00  00 00 00 00 00 00 00 00  |prog_3..........|
000000f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000100  00 00 00 00 00 00 27 10  00 00 00 00 00 00 00 00  |......'.........|
00000110  00 00 00 00 00 00 00 00                           |........        |
```

5 Xbin header, version 001

The “classic” 100-byte, version 000, xbin header starts with the 8 bytes ”RAD-CONT”. The new header format, version 001, starts with the 8 bytes ”RAD-C001”. Subsequent versions will also have version number encoded as RAD-C<*nnn*>. The new header format addresses some problems of the classic header:

1. The new header adds support for multiple programs in a single file.
2. The new format adds parameters needed by EROS (and some utilities), which up to now must have been found with separate programs like `get_rep`, which have got that info by reading the whole xbin binary.
3. The new format allows adding ”metadata” to the header for EROS use. As a first example, there can now be symbolic program names in the header, which EROS can automatically extract when the xbin is loaded by `loadradar`.

The new header is no more of fixed length, but the length info is partly encoded directly to the header itself. This should make it relatively easy to *append* new per-program parameters to the header, without the need to immediately update all programs that make use of the header. That is, on old client programs written for header version N can also read headers of higher version N+m, just by reading only that piece of the header it knows about. The length information in the header allows it to skip over the unknown parts.

Table 13. Master header of xbin version 001. The master header is located in bytes 0–39 of the xbin file. C-code to produce the master header in `tarlan_heat_2.0` is the routine `write_xbin_header()` in the file `/kst/eros5/dsp/tarlan/tarlan_heat_v2/tarlan.y`. For C-routines to interpret the header, see `/kst/eros5/dsp/tarlan/tarlan_heat_v2/xbin_reader.c`.

```
1. Header version RAD–C001                8 bytes – char array.
2. Rc_id5 (like "HOTtx")                  8 bytes – char array.
3. Number (K) of programs in file         4 bytes – big_endian uint32_t.
4. Byte size (N) of each prog header      4 bytes – big_endian uint32_t.
5. Code's byte offset in file             4 bytes – big_endian uint32_t.
6. Total code length in bytes             4 bytes – big_endian uint32_t.
7. Unix time stamp                        4 bytes – big_endian uint34_t.
8. Reserved                               4 bytes – big_endian uint34_t.
```

Table 14. Per-program header of xbin version 001. Each per-program version 001 header is 80 bytes and these are placed in consecutive locations immediately after the master header in the xbin. C-code to produce the per-program header in `tarlan_heat_2.0` is the routine `xbin_header()` in the file `/kst/eros5/dsp/tarlan/tarlan_heat_v2/tarlan.y`. For C-routines to interpret the header, see `/kst/eros5/dsp/tarlan/tarlan_heat_v2/xbin_reader.c`.

```
1. Version                        8 bytes —— char array "RAD–C001"
2. Rc_id5                         8 bytes —— char array "HOTtx" or "HOTrx"
3. Prog size in 8–byte words      4 bytes —— big–endian uint32_t
4. Prog RC start address          4 bytes —— a big–endian uint32_t
5. Prog name                     32 bytes —— char array
6. REP in 100ns units             8 bytes —— big–endian uint64_t
7. RF dutycycle in %              8 bytes —— big–endian double
8. Beam dutycycle in %            8 bytes —— big–endian double
```

6 Compilation speed

Table 15. Compilation speed on an Intel MacBook. As of May 2011, the longest `tlan` file in `/kst/hfexp` was `edmim/357Hz.tlan`, with about 6500 lines of source code, consisting mainly of UPD*, RFON*, RFOFF*, RXSYNC and RXP commands. Compilation results in about 16 000 TX instructions and involves more than 130 000 bit operations on the TX side. The table shows compilation done on a 2 GHz dual core Intel MacBook in 64-bit mode. The first compilation uses the version 2.0 compiler and takes 47 ms total, the latter uses the 1.6 compiler and takes about 19 ms, so the overall speed loss is by a factor of 2.5. According to `top`, the amount of resident memory in the 2.0 case is about 3800 kBytes, and about 700 kBytes in the 1.6 case.

```
macbook:> /kst/eros5/bin/tarlan_heat_2.0 -vv -o 375Hz /kst/hfexp/edmim/357Hz.tlan
Prog 1 (prog_1):      RX sa=0 len=18   bitop=21      TX sa=0 len=16259   bitop=136507
REP= 18 200 000.0 us    RFduty=25.0 %
Parsing: 28.6 ms Codegen: 16.2 ms Filewrite: 1.2 ms Total: 47.0 ms
.rbin => 375Hz.rbin
.tbin => 375Hz.tbin

macbook:> /kst/eros5/bin/tarlan_heat_1.6 -vv -o 375Hz /kst/hfexp/edmim/357Hz.tlan
Warning: no STFIR
Warning: no STC
Warning: no BUFLIP
Prog 1:     RX sa= 0 len= 18     TX sa= 0 len= 16259   REP= 18 200 000.0 us
Parsing: 17.3 ms total: 18.7 ms
.rbin => 375Hz.rbin
.tbin => 375Hz.tbin
```

Table 16. Compilation speed on SOD site's ancient Sun Ultra-250 SPARC. The version 2.0 compilation in this case takes about 540 ms, while with version 1.6 compiler, the time is shorter by a factor of about 3, 170 ms.

```
s2501:> /kst/eros5/bin/tarlan_heat_2.0 -vv 357Hz.tlan
Prog 1 (prog_1):      RX sa=0 len=18   bitop=21      TX sa=0 len=16259   bitop=136507
REP= 18 200 000.0 us    RFduty=25.0 %
Parsing: 258.9 ms Codegen: 275.7 ms Filewrite: 5.1 ms Total: 541.2 ms
.rbin => 357Hz.rbin
.tbin => 357Hz.tbin

s2501:> /kst/eros5/bin/tarlan_heat_1.6 -vv 357Hz.tlan
Warning: no STFIR
Warning: no STC
Warning: no BUFLIP
Prog 1:     RX sa= 0 len= 18     TX sa= 0 len= 16259   REP= 18 200 000.0 us
Parsing: 169.3 ms total: 174.4 ms
.rbin => 357Hz.rbin
.tbin => 357Hz.tbin
```