# Midas BLUE File Format

Version 1.1.1
Released 11-May-2018
$Revision: 327 $

## 1 Preparatory Material

### 1.1 Introduction

The Midas BLUE file is a community standard file format that has evolved over the history of Midas processors to become a de facto standard format for saving data to disk. This volume describes version 1.1 of the Midas BLUE file format and its use in standard Midas processors as controlled by MAINSTRIKE. The Midas frameworks used for these processing systems include X-Midas (with XMPy) and NeXtMidas.

Version 1.0 of the BLUE file format (**BLUE 1.0** see 1.4) was intended to document the existing file format as implemented in Midas frameworks as of early 2006. At that time, part of the processing community moved to use **Platinum** (see 1.4) as the preferred format for file-based interchange. While Platinum is based on BLUE 1.0, and is known as "BLUE 2.0," the Midas processing community still relies upon additional formats, capabilities, and interpretations that have not been adopted by Platinum; as a result, the two formats are not fully interchangeable. This document describes an updated version of the BLUE 1.0 lineage that includes some minor usage updates (including updates to facilitate better compatibility with Platinum) and enhancements. The vast majority of files produced in accordance with this standard will be compatible with BLUE 1.0 and nearly all BLUE 1.0 files are compatible with this standard.

Since 2009, both X-Midas and NeXtMidas have produced files compatible with this document, even though the formal document publication was delayed until 2013.

## 1.2 Document Conventions

Structure field names are represented in `this font`.

Keywords are represented in **THIS FONT**.

Literal text character sequences are represented in **`this font`**.

This document contains tables that define the structure of various portions of the file. The names, types, and short descriptions of the fields are taken from the X-Midas implementation of BLUE files, and may differ slightly in other Midas systems.

Where binary data types are referenced in structures, the type includes a cardinal number that represents the number of bytes used for that data type; for example, the type `int_4` denotes a four-byte (32-bit) signed twos-complement integer, and the type `real_8` denotes an eight-byte (64-bit) floating point number.

## 1.3 Acknowledgments

This document is maintained by MAINSTRIKE. Parts of the document are based upon material provided by IMPACT Science and Technology, Inc.

# Table of Contents

## 1.4 Applicable Documents

**[BLUE 1.1]** *Midas BLUE File Format*, Version 1.1.0, 18 November 2013.

**[Platinum]** *Platinum BLUE Data Exchange Format Standard*, Revision 4 "Volume 1: Platinum BLUE File Format" (`STD-DEV-PLBU/01`), and "Volume 2: Standard Platinum Keywords" (`STD-DEV-PLBU/02`), 16 June 2009.

## 1.5 Time Values

Unless otherwise specified, all time values stored within the BLUE 1.1 file in numeric form are stored in terms of UTC days since 1 January 1950, multiplied by 86,400 seconds/day, plus seconds since UTC

midnight of the current day, excepting leap-seconds (which are not counted).

All measurements in a BLUE 1.1 file given in terms of seconds use the SI second.

*Although slightly obtuse, the above definitions are necessary since the epoch used (1 January 1950) predates the definition of UTC time and because UTC times prior to 1972 were reckoned using a definition of "one UTC second" that was not equal to "one SI second."*

Since leap-seconds are not counted, it is not possible to produce a valid BLUE 1.1 file with a time value that corresponds to the interval during which a leap-second was inserted.

Extended Header Keywords that convey time values and have an accompanying `.UNIT` or `.UNITS` specified as per the Platinum standard may optionally be interpreted in accordance with the Platinum standard; however, any systems providing this option are encouraged to issue a warning in situations where the interpretation using the Platinum standard differs from the interpretation implied by this standard (floating-point rounding errors excepted).

### 1.6 Deprecated Features

In the interest of maximal compatibility with BLUE 1.0 and existing legacy systems, this standard documents a number of "Deprecated" features that may appear under BLUE 1.1, but their use is discouraged on account of limited support within the various Midas baselines.

Systems compatible with this standard are **not** required to support writing of BLUE 1.1 files utilizing Deprecated features and may issue a warning or error upon the reading of a file that utilizes a Deprecated feature.

## 2 History of Midas File Formats

The Midas file format is a community standard format that was introduced nearly 30 years ago as part of the Midas Program. As the years have passed the format has undergone several changes and has been adapted for use by other programs. The original format was Midas GOLD, but this format has been universally superseded by the current Midas BLUE file format.

There are two major families of processing system that use the Midas BLUE format. The X-Midas system, along with its offshoots Midas 2k (in C++), NeXtMidas (in Java), and XMPy (in Python), is the direct descendant of previous Midas systems, and the BLUE format was developed within it.

MARTES is a spin-off from C-Midas, which itself was a conversion of X-Midas' ancestor DCL-MIDAS. MARTES adopted the BLUE file format early on because of its improved portability; however, this early adoption resulted in a divergence of features. Over time, most of the differences have been reconciled, but the MARTES system remains outside the purview of MAINSTRIKE. Material in this document that describes MARTES-specific features or behavior may be particularly inaccurate.

### 2.1 Differences in BLUE File Implementations

### 2.1.1 Extended Header Keywords

Prior to 2005, MARTES used “MARTES ASCII” keywords in the Extended Header. These were phased out between 2002 and 2005 and are only supported under the BLUE 1.0 standard.

### 2.1.2 Additional File Types

Midas BLUE files support several file types to represent a variety of data structures, including one- and two-dimensional data as well as non-homogeneous records. Refer to section 3.1.2 of this volume for a description of BLUE file format types.

X-Midas added support for Type 4000 and Type 5000 files. A Type 4000 file is a series of keyword blocks in the same format as the extended header, and is well-suited to streaming metadata. Type 5000 files are an enhancement to Type 3000 files, adding units and a frame of reference to the subrecords; they were designed for generic system modeling.

MARTES added a Type 6000 file format that has many uses but is most commonly associated with and used for descriptor word files, such as pulse descriptor words (PDWs). Type 6000 files were subsequently adopted into the X-Midas family of systems.

Not all Midas processing systems currently support all file types. In particular, MARTES does not support Type 4000, and Midas 2k does not support Type 6000.

### 2.1.3 Platinum (BLUE 2.0)

The Platinum file format is closely related to Midas BLUE, but is not coupled to the historical implementations used within Midas systems. Some of the Midas BLUE features and requirements were not adopted by Platinum, and Platinum imposes semantics on BLUE keyword interpretation that are not natively recognized by Midas systems. Some of the major differences between Platinum and BLUE include:

- BLUE format Types 4000 and 5000 are not part of Platinum. Specific subtypes of the other major file types are also not common to both systems.
- BLUE does not interpret file contents as a sequence of events. Type 4000 files serve a similar function in BLUE 1.0.
- BLUE uses a strict linear interpolation mechanism to associate time with sample data, while Platinum its event structure to tie precise times to specific samples.

A good overview to the differences between BLUE 1.0 and Platinum, and advice on minimizing incompatibilities between the formats, are provided in the *Guide to Using the Platinum Format*, which can be found in the Platinum documentation on GForge.

# 3 Midas BLUE Format

The file format for Midas BLUE is divided into three regions, as depicted in Figure 1. The format (e.g., byte ordering) for the data portion of the file can be different from that of the header. This allows for modification of header items without necessitating that the entire file be written in the modified format. However, the header control block and the extended header must be in the same format. When changes occur, the processing system writes or rewrites the entire header as necessary to ensure that the combined header blocks are of a uniform format.

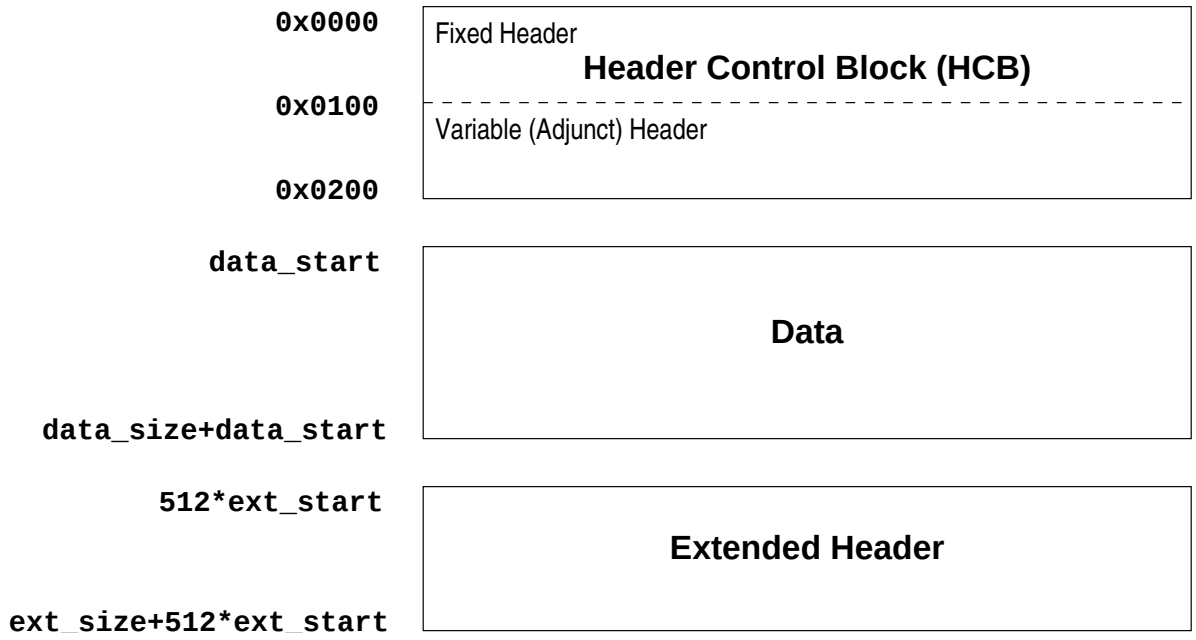


Figure 1: Midas BLUE File Structure

The standard layout of a BLUE file follows the implicit order in Figure 1, and the standard Midas baselines generate most files in this order when written to disk. This format was originally designed to facilitate the collection and processing of real-time data where the Extended Header included information about the data that was not known until processing. By writing the Extended Header after the Data section, the file could be written to disk in a serial fashion.

It is permissible to leave empty space between the HCB and the Data sections, between the Data and Extended Header, and between the Extended Header (or Data section if there is no Extended Header at the end of the file) and the end of the file. This extra space may be done on a mandatory basis (e.g., since the Extended Header is required to start 512-byte boundary, some space is generally required between it and the Data section) or may be done for performance reasons (e.g., X-Midas often will include space following the Extended Header such that the total file size is an integral number of “blocks” in the underlying file system).

Any “space” between the sections of the file will be ignored when reading a file. Systems are not required to zero-fill this space when writing a file, as this permits Midas systems to alter the length of the Data section without altering the Data (e.g., via `HEADERMOD`).

It is also permissible to put the Extended Header earlier in the file than the Data section. All that is required is that the corresponding header fields contain the correct offsets for each section. This is typically done in where some Extended Header fields are required to be written prior to writing the data (as is required to support Type 6000 files within the NeXtMidas pipe system).

BLUE 1.1 files with non-standard layouts will be read correctly by all Midas implementations that support them. However, Midas implementations that modify existing files are not obligated to preserve the original section order, spacing, or data that appears between sections.

## 3.1 Header Control Block (HCB)

As shown in Figure 1, Midas BLUE files are comprised of three blocks of data. The first 512 bytes are referred to as the Header Control Block (HCB). The first 256 bytes of the HCB are the *fixed* portion of the HCB header and the remaining 256 bytes are the *variable* portion of the HCB header. Each is described below.

The BLUE header structure is used within Midas processing systems as well as in files. Some fields in the structure have meaning only within those systems. Other systems should not make assumptions about the values held in these fields when operating on BLUE files. In support of future extensions that may use these fields to convey additional information, systems that create BLUE files should set the fields to zero. Systems that manipulate existing BLUE files should reset values of undefined fields to zero in case a previous value was invalidated as a result of the manipulations.

### 3.1.1 Fixed Header

The fixed header portion of the HCB is comprised of the fields in Table 2. Sections 3.1.1.1 through 3.1.1.25 provide further details for each field.

Table 2: BLUE Header Fields

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 0 | `version` | 4 | `char[4]` | Header version |
| 4 | `head_rep` | 4 | `char[4]` | Header representation |
| 8 | `data_rep` | 4 | `char[4]` | Data representation |
| 12 | `detached` | 4 | `int_4` | Detached header |
| 16 | `protected` | 4 | `int_4` | Protected from overwrite |
| 20 | `pipe` | 4 | `int_4` | Pipe mode (N/A) |
| 24 | `ext_start` | 4 | `int_4` | Extended header start, in 512-byte blocks |
| 28 | `ext_size` | 4 | `int_4` | Extended header size in bytes |
| 32 | `data_start` | 8 | `real_8` | Data start in bytes |
| 40 | `data_size` | 8 | `real_8` | Data size in bytes |
| 48 | `type` | 4 | `int_4` | File type code |

... continued on next page

Table 2: BLUE Header Fields (... continued)

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 52 | format | 2 | char[2] | Data format code |
| 54 | flagmask | 2 | int_2 | 16-bit flagmask (1=flagbit) |
| 56 | timecode | 8 | real_8 | Time code field |
| 64 | inlet | 2 | int_2 | Inlet owner |
| 66 | outlets | 2 | int_2 | Number of outlets |
| 68 | outmask | 4 | int_4 | Outlet async mask |
| 72 | pipeloc | 4 | int_4 | Pipe location |
| 76 | pipesize | 4 | int_4 | Pipe size in bytes |
| 80 | in_byte | 8 | real_8 | Next input byte |
| 88 | out_byte | 8 | real_8 | Next out byte (cumulative) |
| 96 | outbytes | 64 | real_8[8] | Next out byte (each outlet) |
| 160 | keylength | 4 | int_4 | Length of keyword string |
| 164 | keywords | 92 | char[92] | User defined keyword string |
| 256 | adjunct | 256 | char[256] | Type-specific adjunct union |

**3.1.1.1 `version`** The `version` field indicates the Midas file type as a four-character string. The value stored in the field should be **`BLUE`**. The only other format that can occur here is the legacy **`GOLD`** format, which is no longer supported in most Midas systems.

**3.1.1.2 `head_rep`** The `head_rep` (header representation) field identifies the byte ordering for information stored in the header: specifically in the HCB structure (including adjunct structures), and in the extended header. The valid values for this field are shown in Table 3. Values less than four characters long should be padded with spaces to fill the four character field.

Table 3: Valid Values for Header and Data Representation Fields

| Code | Byte Order | Floating Point Representation |
|---|---|---|
| **`IEEE`** | Big endian | IEEE 754 |
| **`EEEI`** | Little endian | IEEE 754 |
| **`VAX`** | Little endian | VAX *(Deprecated)* |
| **`CRAY`** | Big endian | CRAY *(Deprecated)* |

Nearly all modern processors use IEEE 754 floating-point values (though the choice of big-endian (**`IEEE`**)

or little-endian (**EEEI**) byte ordering varies). Existing legacy storage formats (**VAX** and **CRAY**) may occasionally be encountered, but their use is considered Deprecated since they lack hardware support in modern computer architectures and have limited support in the Midas baselines (e.g., NeXtMidas does not support reading **CRAY** floating-point values).

**3.1.1.3 data_rep** The data_rep (data representation) field identifies the byte ordering for information in the data portion of the file, using the codes defined in Table 3. There is no requirement that head_rep and data_rep be the same within a given BLUE file. This variability allows a system to annotate or modify the keywords of a file without having to convert the data itself.

**3.1.1.4 detached** The detached flag is set to a non-zero value to indicate that the data itself exists in another file. In this situation, the term *Midas BLUE file* refers to a file that contains only and all the header and extended keyword information. If the BLUE file is named filename.tmp or filename.prm, the corresponding data must be in a separate file, generally collocated with the header and named filename.det. Absent other instruction, a non-zero value in the detached field is should be interpreted this way. However, specific Midas systems sometimes use other values to convey alternative location information, as detailed in the next following subsections.

**3.1.1.4.1 X-Midas detached Semantics** A value from 2 through 127 indicates that the data is located in a file filename.det that can be found in the X-Midas auxiliary path identified by the value.

**3.1.1.4.2 NeXtMidas detached Semantics** A value greater than 128 indicates that the extension for the data file is not .det, but is instead defined by re-interpreting the byte sequence encoding the detached field as though it were a char[4] field.

**3.1.1.5 protected** The protected flag is a soft setting that instructs programs/primitives not to allow a file to be edited. When the protected flag is set to a non-zero value, the file should be maintained as read-only; however, given that this flag is a soft setting, users do have the ability to modify the software to allow for modification of the file. Systems producing BLUE files should set the field to either zero or one.

**3.1.1.6 pipe** The pipe field is used in X-Midas pipes and is not applicable to BLUE files. Systems producing BLUE files should set the field to zero.

**3.1.1.7 ext_start** The ext_start (extended header starting offset) field indicates the number of 512 byte blocks from the start of the file to the start of the extended header. For example, a value of 1000 would indicate that the extended header can be located starting at the 512000th byte of the file. Note: The

type and units of this field impose a size restriction of one terabyte on the data section of any non-detached file that includes an extended header. If the `ext_start` field is set to zero, the `ext_size` field must also be set to zero; this indicates the absence of any extended header.

**3.1.1.8 `ext_size`** The `ext_size` (extended header size) field indicates the size, in bytes, of the extended header. If the `ext_size` field is set to zero, the `ext_start` field must also be set to zero; this indicates the absence of any extended header.

**3.1.1.9 `data_start`** The `data_start` indicates the offset of the data from the start of the file, in bytes. The offset should be a multiple of 512 bytes since historically data has been written in 512-byte blocks. Detached files typically start either at 0 (zero) or at a file-appropriate location (e.g., after a non-Midas header). The `data_start` field in non-detached files is typically 512 to indicate that the data immediately follows the HCB, but may be larger in some circumstances. For example, both Cray and IEP cdif files use a `data_start` of 4096. Since files can have detached data, this field has the potential to overlap with the locations indicated by the `ext_start` given that those locations are not within the Midas BLUE file. Depending upon the location of the data (as indicated by the `data_start` field) and the location of the extended header (as indicated by the `ext_start` field), there can be gaps in the Midas BLUE file (i.e., places where no information is written). It should not be assumed that a Midas processing system will preserve those gaps when modifying a BLUE file.

**3.1.1.10 `data_size`** The `data_size` field indicates the size, in bytes, of the data portion of the file.

**3.1.1.11 `type`** The `type` field indicates the type of data contained in the file. The type value divided by 1000 indicates the overall file structure; the value modulo 1000 indicates that the file contains a particular kind of data. The file types are described in detail in the section 3.1.2 in the context of the structure of each type's adjunct header. Allowable and common special-use file types are described in Table 4.

Table 4: Midas File Types

| Type | Contents |
|---|---|
| 1000 | Generic one-dimensional scalar data |
| 1001 | Uniformly sampled amplitude data |
| 1002 | Time-of-arrival (TOA) data |
| 1003 | Histogram data |
| 1004 | Uniformly sampled asynchronous burst data |
| 1005 | Multi-point data |
| 1200 | Packetized one-dimensional scalar data |
| 1998 | Records of connected points for filled polygons |

... continued on next page

Table 4: Midas File Types (... continued)

| Type | Contents |
|---|---|
| 1999 | Records of connected points (e.g., mapping polygons or complex data) |
| 2000 | Generic two-dimensional scalar data |
| 2001 | Frame data (e.g., spectra vs. time) |
| 2004 | Burst with uniform intra- and inter-burst sampling |
| 2200 | Packetized two-dimensional scalar data |
| 3000 | Record structured data (non-homogeneous, limited to 26 subrecords) |
| 3999 | Records indicating a set of parameters |
| 4000 | Key/value pairs |
| 5000 | Record structured data (non-homogeneous, limited to 14 components) plus system modeling parameters |
| 5001 | State vector information |
| 5010 | Geodetic reference positions |
| 6000 | Record structured data (non-homogeneous) |
| 6001 | Uniformly sampled record data |
| 6002 | Multi-point record data |
| 6003 | Time-based data |
| 6004 | Time-based data associated with a Type 1004 burst file |
| 6005 | Index data associated with a Type 1004 multi-point file |
| 6080 | SDDS network packet data (56-byte SDDS header, 1024-byte data) |

**3.1.1.12 `format`** The `format` field specifies the format of the data itself via a 2-character designator. For most uses, the format is an encoded digraph specifying the size (first character) and underlying type (second character) of a data element. The designator takes on other values in situations where the size/type encoding is not appropriate; see Table 7.

**3.1.1.12.1 Format Size Codes** As an encoded digraph, the first character is a size or multiplier code that indicates the number of atomic elements contained within a single data element. The size designators most commonly seen include codes **`S`**, **`C`**, **`V`**, **`1`** through **`9`**, and **`X`**. Size designator codes are presented in Table 5.

Table 5: Data Format Size Designator

| Code | Definition | Comments |
|---|---|---|
| **`S`** | Scalar | 1 element per data point |

... continued on next page

Table 5: Data Format Size Designator (... continued)

| Code | Definition | Comments |
|---|---|---|
| **C** | Complex | 2 elements per data point, representing a complex number (real, imaginary) |
| **V** | Vector | 3 elements per data point, representing a positional vector (x,y,z) or (alt,lat,lon) |
| **Q** | Quad | 4 elements per data point, representing a positional vector with time (x,y,z,time) |
| **M** | Matrix | 9 elements per data point, representing a 3-by-3 coordinate transformation |
| **T** | Transform Matrix | 16 elements per data point, representing a 4-by-4 coordinate transformation with scaling *(may be used as a generic multiplier for ASCII strings)* |
| **U** | User Defined | **Deprecated** Extremely limited support in Midas baselines. |
| **1 - 9** | Generic multiplier | 1 to 9 elements per data point |
| **X** | Generic multiplier | 10 elements per data point |
| **A** | Generic multiplier | 32 elements per data point |

Although the generic multipliers overlap the non-generic forms in terms of number of "elements per data point" (e.g., **2** and **C** both convey "2 elements per data point"), many processing systems handle them differently. For example, many Midas commands that take in a complex-valued input specifically check for the format size designator **C** and presume that a file with format size designator **2** does not represent complex data. In general, all ASCII strings should use a generic multiplier (or **T**).

The **A** Format Size Designator is not included in the BLUE 1.0 standard and was added in 2009 to support longer text fields within files (e.g., AA permits a 256-character string) and for compatibility with Platinum.

**3.1.1.12.2 Format Type Codes** The second character of the encoded digraph is the atomic data type of the element. It specifies the actual data representation of each data element. There are eight atomic data types currently defined for numeric data, plus one for character data. The atomic data types are listed in Table 6.

Table 6: Data Format Type Designator

| Code | Definition | Supported In |
|---|---|---|
| **B** | 8-bit integer | KW, DATA, SUBRECORDS |
| **I** | 16-bit integer | KW, DATA, SUBRECORDS |

... continued on next page

Table 6: Data Format Type Designator (... continued)

| Code | Definition | Supported In |
|---|---|---|
| **L** | 32-bit integer | KW, DATA, SUBRECORDS |
| **X** | 64-bit integer | KW, DATA, SUBRECORDS |
| **F** | 32-bit float | KW, DATA, SUBRECORDS |
| **D** | 64-bit float | KW, DATA, SUBRECORDS |
| **P** | Packed bits (accessible on byte boundaries) | DATA, SUBRECORDS |
| **N** | 4-bit integer (accessible on byte boundaries) | DATA, SUBRECORDS |
| **O** | Offset byte | DATA, SUBRECORDS |
| **A** | ASCII (8 characters) | DATA, SUBRECORDS |
| **A** | ASCII (variable length) | KW |
| **S** | Modified UTF-8 (8 octets) | Reserved for future use in DATA, SUBRECORDS |
| **S** | Modified UTF-8 (variable length) | Reserved for future use in KW |
| **T** | 32-bit integer | KW |

In BLUE version 1.1, all numerical data types are signed. This is specifically chosen due to the limited support for unsigned numbers in a number of programming languages including Fortran, Java, and Python (the three of which play key roles in the various Midas baselines). Non-Midas implementations, such as MARTES, have used **U** to represent "16-bit unsigned integer" and **V** to represent "32-bit unsigned integer", but these values are unrecognized by the Midas frameworks.

All integral format types except **O** use twos-complement representations. Offset byte, also known as offset binary, is used by various analog-to-digital conversion hardware to represent analog values between positive and negative full scale. This format is not supported in keywords. Values may be converted from **O** format to **B** format by interpreting the **O** value as an unsigned eight-bit integer and subtracting 128. *(Note that :lit:'O' format is not permitted in SUBRECORDS under the Platinum standard.)*

**X** format support was added into X-Midas during the version 4.3 release series, and NeXtMidas in version 1.8.0. Many legacy Midas processing systems will be unable to read files that have data or keywords with **X** type data elements.

The **T** keyword code was used in versions of X-Midas prior to 4.9.0 to represent a widget's numeric tag value when such result parameters are stored in a file. This behavior is now considered deprecated and should be avoided. Results with type **T** are now stored using the **L** code.

Sample counts for packed bit data reflect the individual bits; the byte count will be correspondingly smaller. Bits are packed into bytes starting with the most significant bit (MSB0): `mask = 0x80 >> (idx mod 8)`; e.g., a sequence of four one-valued bits would be represented as 0xF0.

For historical reasons, NeXtMidas packs bits into bytes starting with the least significant bit (LSB0): `mask`

`= 0x01 << (idx mod 8)`. It also uses the byte order field as an indicator of whether bit ordering is LSB0 (**`IEEE`**) or MSB0 (**`EEEI`**). When processing a file with `format` = **`SP`**, it will internally replace the file's `data_rep` field with **`IEEE`**, and when run on a big-endian (**`EEEI`**) host will reverse the order of bits in packed data. The net effect is that SP data passing through NeXtMidas may be misinterpreted in various ways that depend on file and host byte ordering.

The **A** code has slightly different semantics depending on context: in keyword values, it represents a variable-length string, not necessarily an eight character sequence. When **A** is used to indicate a fixed-length ASCII field, values should be padded with ASCII space characters to their full length. Although **A** permits use of any ASCII characters, the use of non-printing characters (other than the ASCII space) is strongly discouraged as there is inconsistent support for them in the Midas baselines.

The **S** format is reserved for future use to indicate a Modified UTF-8 string. No Midas baselines currently support **S**, but there have been multiple requests to add UNICODE support in the future. If UNICODE support is added in the future, it will use the format code **S** and will be encoded using Modified UTF-8, which is defined as follows:

```
Modified UTF-8 format is identical to "normal" UTF-8 except that any
embedded NUL values within the string are encoded as two-byte values
(similar to what is done in the Java VM). Note that unlike the Java
VM and some other 16-bit UTF representations in C++ this does NOT
rely on "surrogate pairs" and supports the 1-/2-/3-/4-octet forms of
UTF-8.
  Values in the 0x0001 to 0x007F range (ASCII, except NUL):
    Octet 1: [ 0 x x x x x x x ]

  Values in the 0x0080 to 0x07FF range and 0x0000 (ASCII NUL):
    Octet 1: [ 1 1 0 x x x x x ]
    Octet 2: [ 1 0 x x x x x x ]

  Values in the 0x0800 to 0xFFFF range:
    Octet 1: [ 1 1 1 0 x x x x ]
    Octet 2: [ 1 0 x x x x x x ]
    Octet 3: [ 1 0 x x x x x x ]

  Values in the 0x10000 to 0x10FFFF range:
    Octet 1: [ 1 1 1 1 0 x x x ]
    Octet 2: [ 1 0 x x x x x x ]
    Octet 3: [ 1 0 x x x x x x ]
    Octet 4: [ 1 0 x x x x x x ]
```

When **S** is used to indicate a fixed-length UTF-8 field, values should be padded with zeros (0x00) to their full length. This usage permits string length to be determined by existing C/C++ library functions that search for the first ASCII NUL (0x00) value.

"Nibbles" are used in NeXtMidas and are commonly encountered as packetized output from 4-bit digitizers; they are essentially used to pack two distinct 4-bit integers into a single byte (8-bit).

Since most CPUs and programming languages do not support a 4-bit integer type, nibbles are typically processed as bytes while in memory and are only compacted for transmission or when saving to a file. When a (byte) data value is converted/packed into Nibble data, it is assumed to already be normalized (into the lower [0-3] bits) so the higher [4-7] bits are discarded by truncation (this is similar to how 32-bit values are converted into 16-bit values in Java).

Two Nibbles are packed into one byte in a BLUE file with the first nibble in the lower bits [0-3] and the second nibble in the higher bits [4-7]. For example, the following four nibbles (0x1, 0x2, 0x3, 0x4) get packed into two bytes as (0x21, 0x43).

Where the size of a file indicates and odd number of nibble values, an additional nibble value must be inserted at the end of the file (or end of the subrecord) to pad the file (or subrecord) out to an even number of nibble values. When the file is read in, this extra space will be ignored. For example, a file of size 999 ending with nibble values (0x7, 0x8, 0x9) may have the last two bytes written as (0x87, 0x09); although the use of 0 as a pad value is encouraged, it is not required.

**3.1.1.12.3 Other Format Digraphs** In addition to type codes as described above, the format designator for Type 3000, 4000, 5000, and 6000 files may be set to codes defined in Table 7. For these file types, the format designator does not affect the interpretation of the file structure, which is instead provided through information stored in the type-specific adjunct header. See section 3.1.2.

Table 7: Additional Format Designators

| Code | Definition | Comments |
|---|---|---|
| **KW** | Keywords | For Type 4000 files, indicating contents in keyword format |
| **NH** | Non-homogeneous | For Type 3000, 5000, 6000 record-oriented files |

Early versions of the Platinum standard described format digraphs RB and RE -- these values are deprecated in current versions of the Platinum standard and are not permitted in any BLUE 1.0 or BLUE 1.1 files.

**3.1.1.13 `flagmask`** The `flagmask` field is used in X-Midas pipes and is not applicable to BLUE files. Systems producing BLUE files should set the field to zero.

**3.1.1.14 `timecode`** The `timecode` field specifies the epoch time for file data, in seconds since 1 January 1950 (see 1.5 for details). This epoch is the zero point for the first sample in the file (`xstart` for Type 1000, `ystart` for Type 2000, `rstart` for Types 3000 and 6000, and `tstart` for Type 5000). It is stored in a single 64-bit floating point number, representing the epoch time down to microsecond resolution. Precision time with sub-microsecond resolution (down to the picosecond) for the first sample can be obtained by adding the appropriate `xstart` value to `timecode` and adjusting by the value of the main header keyword **TC_PREC**; see section 3.4.1.5. For example, the start time for a Type 1000 file is:

start_time = `timecode` + **TC_PREC** + `xstart`

Changes to **TC_PREC** may be required if the value of `timecode` is ever changed.

**3.1.1.15 `inlet`** The `inlet` field is used in X-Midas pipes and is not applicable to BLUE files. Systems producing BLUE files should set the field to zero.

**3.1.1.16 `outlets`** The `outlets` field is used in X-Midas pipes and is not applicable to BLUE files. Systems producing BLUE files should set the field to zero.

**3.1.1.17 `outmask`** The `outmask` field is used in X-Midas pipes and is not applicable to BLUE files. Systems producing BLUE files should set the field to zero.

**3.1.1.18 `pipeloc`** The `pipeloc` field is used in X-Midas pipes and is not applicable to BLUE files. Systems producing BLUE files should set the field to zero.

**3.1.1.19 `pipesize`** The `pipesize` field is used in X-Midas pipes and is not applicable to BLUE files. Systems producing BLUE files should set the field to zero.

**3.1.1.20 `in_byte`** The `in_byte` field is used in X-Midas pipes and is not applicable to BLUE files. Systems producing BLUE files should set the field to zero.

**3.1.1.21 `out_byte`** The `out_byte` field is used in X-Midas pipes and is not applicable to BLUE files. Systems producing BLUE files should set the field to zero.

**3.1.1.22 `out_bytes`** The `out_bytes` field is used in X-Midas pipes and is not applicable to BLUE files. Systems producing BLUE files should set the field to zero.

**3.1.1.23 `keylength`** The `keylength` field specifies the valid length, in bytes, of the keyword section located in the main header.

**3.1.1.24 `keywords`** The `keywords` field stores keywords in the main header. Certain keywords should always be placed in the main header so they can be quickly located by systems that process BLUE files. See 3.4.1 for details.

**3.1.1.24.1 Format For Main Header Keywords** The format for main header keywords is a key/value pairs, each pair terminated with an ASCII end-of-string NUL character, and the key and value separated by an equals sign, as shown in Table 8. Keyword values within the header control block must use this ASCII representation.

Table 8: Main Header Keyword Format

| Content | Type | Comments |
|---|---|---|
| *tag* | `char[]` | Keyword tag (name) |
| **=** | `char` | ASCII equals sign (decimal value 61) |
| *value* | `char[]` | Keyword value |
| 0 | `int_1` | ASCII NUL (decimal value 0) |

Although *tag* may include any ASCII character, other than NUL, use of lower-case letters and non-printing characters are strongly discouraged as there is limited support for them in the current Midas baselines.

**3.1.1.25 `adjunct`** The contents of the second half of the header control block provide structural information related to specific types of BLUE files. These structures are described in section 3.1.2.

### 3.1.2 Variable (Adjunct) Header

The structure of the variable portion of the Header Control Block depends on the file type specified in the fixed portion of the HCB (refer to Table 4 for a list of file types). The variable portion of the HCB is 256 bytes long. The description for the variable portion of the HCB for each of the file types is provided below. Offsets in the structural table are relative to the start of the adjunct field. Where specific file types require the presence of specific keywords to determine the structure of the file content, those keywords are described in the corresponding file-type section.

**3.1.2.1 Type 1000** Type 1000 files contain a series of homogeneous samples. Normally used to store digitized data, this format can also be used to store any type of generic, one-dimensional, homogeneous data. The structure of the Type 1000 adjunct region is described in Table 9.

#### 3.1.2.1.1 Adjunct Structure

Table 9: Type 1000 Adjunct Structure

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 0 | xstart | 8 | real_8 | Abscissa value for first sample |
| 8 | xdelta | 8 | real_8 | Abscissa interval between samples |
| 16 | xunits | 4 | int_4 | Units for abscissa values |

**3.1.2.1.1.1 `xstart`** The `xstart` field specifies, in units identified by `xunits`, the start time of the first sample of the data, relative to the file epoch time as defined by the `timecode` field and **TC_PREC** keyword. See section 3.4.1.5.

**3.1.2.1.1.2 `xdelta`** The `xdelta` field specifies, in units identified by `xunits`, the interval between data samples contained within the file. The `xdelta` field is commonly referred to as the abscissa, which is 1/sample rate.

**3.1.2.1.1.3 `xunits`** The `xunits` field specifies units of measure for both the `xstart` and `xdelta` fields. See Table 29.

**3.1.2.1.2 Type 1998 and 1999 Files** It is common to use a Type 1998 and 1999 file with complex data to represent plottable map objects, where the real component represents an X axis value or a longitude, and the imaginary component represents a Y axis value or a latitude. Type 1998/9 files may also represent variable-length records, where the initial complex numbers in the record specify its length and other plot characteristics such as symbol color or bounding box information. The Explain pages for the X-Midas MAPMAN command provide the standard interpretation of these files for the XGEOPLOT map plotter.

**3.1.2.2 Type 2000** Type 2000 files contain a series of homogeneous records, all of the same length, generally representing framed data. These files can be the output of an FFT, video frame data, or any other two-dimensional homogeneous data. The format of each frame element is specified by the HCB format field.

**3.1.2.2.1 Adjunct Structure**

Table 10: Type 2000 Adjunct Structure

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 0 | xstart | 8 | real_8 | Frame (row) starting value |
| 8 | xdelta | 8 | real_8 | Increment between samples in frame |
| 16 | xunits | 4 | int_4 | Frame (row) units |
| 20 | subsize | 4 | int_4 | Number of data points per frame (row) |
| 24 | ystart | 8 | real_8 | Abscissa (column) start |
| 32 | ydelta | 8 | real_8 | Increment between frames |
| 36 | yunits | 4 | int_4 | Abscissa (column) unit code |

**3.1.2.2.1.1 `xstart`** The `xstart` field specifies an abscissa-style value associated with the first element in each frame. For example, in Type 2000 files storing a series of one-dimensional FFT results each frame represents a frequency interval, with `xstart` specifying the frequency associated with the lower end of the interval. For files with real-valued samples `xstart` is typically zero, while for files with complex-valued samples `xstart` is typically `bw/2`.

**3.1.2.2.1.2 `xdelta`** The `xdelta` field specifies the interval between consecutive samples within a frame.

**3.1.2.2.1.3 `xunits`** The `xunits` field specifies the units associated with the `xstart` and `xdelta` values. See Table 29.

**3.1.2.2.1.4 `subsize`** The `subsize` field specifies the number of data elements in each frame; i.e., the row length.

**3.1.2.2.1.5 `ystart`** The `ystart` field in a Type 2000 file is interpreted in the same way as the `xstart` field in a Type 1000 file, except that it refers to the start time of the first frame. See section 3.1.2.1.2.

**3.1.2.2.1.6 `ydelta`** The `ydelta` field in a Type 2000 file is interpreted in the same way as the `xdelta` field in a Type 1000 file, except that it defines the interval between consecutive data frames in the file. See section 3.1.2.1.3.

**3.1.2.2.1.7 `yunits`** The `yunits` field specifies the units associated with the `ystart` and `ydelta` values. See Table 29.

**3.1.2.3 Type 3000** Type 3000 files contain a series of non-homogeneous records, all with the same structure. The most common usage is with time-oriented records, where by convention the time is stored as the first column or field of the record. The HCB format field for Type 3000 files should be set to **NH**, except in the rare case that all columns of the record have the same format, in which case either **NH** or the format common to the columns may be used. Setting the field to something other than **NH** is useful only because it allows a standard Midas primitive (headermod) to change the file to a Type 2000 file. Most Midas primitives ignore the format field in Type 3000 files.

#### 3.1.2.3.1 Adjunct Structure

Table 11: Type 3000(6000) Adjunct Structure

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 0 | rstart | 8 | real_8 | abscissa value for first record |
| 8 | rdelta | 8 | real_8 | abscissa distance between records |
| 16 | runits | 4 | int_4 | units for record abscissa values |
| 20 | subrecords | 4 | int_4 | number of columns per record (max of 26) |
| 24 | r2start | 8 | real_8 | abscissa value for first column in record |
| 32 | r2delta | 8 | real_8 | abscissa value between record columns |
| 40 | r2units | 4 | int_4 | units for column abscissa values |
| 44 | record_length | 4 | int_4 | length of record in bytes |
| 48 | subr | 208 | SUBRECSTRUCT[26] | record column definitions |

**3.1.2.3.1.1 `rstart`** The rstart field in a Type 3000 file is interpreted in the same way as the xstart field in a Type 1000 file, except that it refers to the start time of the first record. See section 3.1.2.1.2. When the time associated with a record is stored within the record itself, this field is generally set to 0.

**3.1.2.3.1.2 `rdelta`** The rdelta field in a Type 3000 file is interpreted in the same way as the xdelta field in a Type 1000 file, except that it defines the interval between consecutive records in the file. See section *xdelta*. When the time associated with a record is stored within the record itself, this field is generally set to 1.

**3.1.2.3.1.3 `runits`** The runits field in a Type 3000 file is interpreted in the same way as the xunits field in a Type 1000 file, except that it specifies units for the rstart and rdelta fields. See section 3.1.2.1.4.

**3.1.2.3.1.4 `subrecords`** The `subrecords` field indicates the number of columns or fields in each record within the file. The storage space associated with the `subr` field limits the value to no more than 26.

**3.1.2.3.1.5 `r2start`** The `r2start` field specifies a value associated with the first column or field of each record. This is generally set to 0.

**3.1.2.3.1.6 `r2delta`** The `r2delta` field specifies an interval between columns or fields of each record. This field is generally set to 1.

**3.1.2.3.1.7 `r2units`** The `r2units` field exists only for structural compatibility with other adjunct structures, and is not used in Type 3000 files. The field is normally set to zero.

**3.1.2.3.1.8 `record_length`** The `record_length` field defines the length of each record in bytes. Due to padding requirements, this may be larger than the sum of the lengths of the record columns.

**3.1.2.3.1.9 `subr`** The `subr` field contains a sequence of record column or field definitions, each of which is an eight-byte structure as defined in section 3.1.2.3.12. Systems intending to process Type 3000 files should take into account that the column definitions in the `subr` field may not be presented in strictly increasing offset order.

**3.1.2.3.2 Extended Type 3000 Files** Extended Type 3000 files are used operationally, though their use is not encouraged. These files are similar to a standard Type 3000 file, except that they are not limited to 26 `subrecords`. Increasing the number of `subrecords` is done by extending the size of the adjunct header beyond 256 bytes, as shown below. This extension of the adjunct header necessitates the use of a `data_start` that is greater than 512.

Table 12: Extended Type 3000(6000) Adjunct Structure

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 0 | `rstart` | 8 | `real_8` | *no change* |
| 8 | `rdelta` | 8 | `real_8` | *no change* |
| 16 | `runits` | 4 | `int_4` | *no change* |
| 20 | `subrecords` | 4 | `int_4` | number of columns per record (no max limit) |
| 24 | `r2start` | 8 | `real_8` | *no change* |

... continued on next page

Table 12: Extended Type 3000(6000) Adjunct Structure (... continued)

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 32 | r2delta | 8 | real_8 | *no change* |
| 40 | r2units | 4 | int_4 | *no change* |
| 44 | record_length | 4 | int_4 | *no change* |
| 48 | subr | N*8 | SUBRECSTRUCT[N] | record column definitions (where N = MAX(26, subrecords)) |

Although supported by NeXtMidas for over a decade, Extended Type 3000 files have limited support in the other Midas baselines.

**3.1.2.3.3 Type 3000 SUBRECSTRUCT Structures** Each sub-record is a sequence of column or field values. The columns are defined using the structure in Table 13.

Table 13: Type 3000 SUBRECSTRUCT Structure

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 0 | name | 4 | char[4] | column name |
| 4 | format | 2 | char[2] | format for column data |
| 6 | offset | 2 | int_2 | offset within record for column start |

**3.1.2.3.3.1 name** The name field associates a title or name with the corresponding column or field of the record. The value should be padded with spaces to four characters.

Although name may include any ASCII character, use of lower-case letters and non-printing characters (other than spaces to pad out names shorter than four characters) are strongly discouraged as there is limited support for them in the current Midas baselines.

**3.1.2.3.3.2 format** The format field identifies the format of data within the column, using the digraph encoding described in section 3.1.1.12.1.

**3.1.2.3.3.3 offset** The offset field specifies the offset, in bytes, of the start of the column data, relative to the start of the record.

**3.1.2.3.4 Type 3999 Files** Type 3999 files are a legacy variant where multiple adjacent records are combined into a single record. The functionality expressed by Type 3999 files has been replaced by the capabilities of Type 4000 files, and Type 3999 files should not be encountered in normal Midas systems. For details on the format of Type 3999 files, see the UCL option tree in the X-Midas distribution.

**3.1.2.4 Type 4000** Type 4000 files contain a series of heterogeneous records of varying structure. Each record is represented as a sequence of keywords, with the same structure as X-Midas binary extended header keywords; see section 3.3.1.

#### 3.1.2.4.1 Adjunct Structure

Table 14: Type 4000 Adjunct Structure

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 0 | vrstart | 8 | real_8 | unused |
| 8 | vrdelta | 8 | real_8 | unused |
| 16 | vrunits | 4 | int_4 | unused |
| 20 | nrecords | 4 | int_4 | number of variable-length records in file |
| 24 | vr2start | 8 | real_8 | unused |
| 32 | vr2delta | 8 | real_8 | unused |
| 40 | vr2units | 4 | int_4 | unused |
| 44 | vrecord_length | 4 | int_4 | fixed length |

**3.1.2.4.1.1 `vrstart`** The `vrstart` field exists only for structural compatibility with other adjunct structures, and is not used in Type 4000 files. The field is normally set to zero.

**3.1.2.4.1.2 `vrdelta`** The `vrdelta` field exists only for structural compatibility with other adjunct structures, and is not used in Type 4000 files. The field is normally set to zero.

**3.1.2.4.1.3 `vrunits`** The `vrunits` field exists only for structural compatibility with other adjunct structures, and is not used in Type 4000 files. The field is normally set to zero.

**3.1.2.4.1.4 `nrecords`** The `nrecords` field specifies the number of records (keyword sets) in the file. For fixed-length records, this value may be zero or negative, and the number of records is calculated from the file size. For variable-length records, this value must be a correct, positive, value. See section 3.1.2.4.9.

**3.1.2.4.1.5 `vr2start`** The `vr2start` field exists only for structural compatibility with other adjunct structures, and is not used in Type 4000 files. The field is normally set to zero.

**3.1.2.4.1.6 `vr2delta`** The `vr2delta` field exists only for structural compatibility with other adjunct structures, and is not used in Type 4000 files. The field is normally set to zero.

**3.1.2.4.1.7 `vr2units`** The `vr2units` field exists only for structural compatibility with other adjunct structures, and is not used in Type 4000 files. The field is normally set to zero.

**3.1.2.4.1.8 `vrecord_length`** The `vrecord_length` field is used to describe the length of each record. If the value of this field is positive, each record in the file has this length. The total number of records in the file is determined from the file size and the record length.

If `vrecord_length` is zero or negative, records have varying lengths, and the `nrecords` field must be valid. If `vrecord_length` is zero, no index exists for the file and the records must be examined in sequence from the start of the file. If `vrecord_length` is negative, the file is a keyword indexed file, and the extended header must contain a keyword **T4INDEX**, the value of which must be an `nrecords`-element array of file offsets as described in section 3.1.2.4.15.

**3.1.2.4.2 Variable Record Structure** Each record begins with an eight byte header followed by the record, expressed as a series of keywords.

Table 15: Type 4000 Variable Record Structure

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 0 | `fsize` | 4 | `int_4` | length of record |
| 4 | `size` | 4 | `int_4` | length of data in record |
| 8 | `data` | `fsize` | `int_1` | data |
| 8 + `size` | *padding* | `fsize - size` | `int_1` | padding |

**3.1.2.4.2.1 `fsize`** The `fsize` field specifies the length of the record or keyword set, including any trailing padding but not including the eight-byte header.

**3.1.2.4.2.2 `size`** The `size` field specifies the length of the data portion of the record, exclusive of the header and any trailing padding.

**3.1.2.4.2.3 `data`** A sequence of one or more keywords in binary format as documented in section 3.3.1. Each record can be seen as an independent extended header.

#### 3.1.2.4.3 Keywords

**3.1.2.4.3.1 T4INDEX** For a keyword indexed Type 4000 file, the extended header keyword **T4INDEX** must have a value that is an `nrecords`-element array of double-precision floating point numbers, each of which represents the offset of the corresponding keyword block, in bytes from file start (not relative to the start of the data region).

**3.1.2.4.4 Known Non-Standard Extensions** Since the type indicator for a keyword value can specify only the underlying data format type (cf. Table 6), and not a data format size (cf. Table 5), application writers have used the HCB `format` field to indicate the data format of one particular keyword they are transporting in a Type 4000 record (typically named **`DATA`**). This is only a convention observed; to be truly useful, some sort of aggregate format ability should be added to Type 4000 and regular extended header keywords.

**3.1.2.5 Type 5000** Type 5000 files were designed in the mid 1980s to store information about motion in a frame of reference. They contain a series of non-homogeneous records, all with the same structure. They differ from Type 3000 files in that some of the adjunct header is used to store additional parameters that apply to the entire file, and by associating a units tag with each record field. The most common usage is for state vectors and geodetic positions.

The HCB `format` field for Type 5000 files is set to **`NH`** (Non-Homogeneous) except in the rare case that all columns of the record have the same format, in which case either **`NH`** or the format common to the columns may be used. (Since most Type 5001 files include only `VD` fields, it is common to see Type 5001 files with the `format` listed as `VD`.)

#### 3.1.2.5.1 Adjunct Structure

Table 16: Type 5000 Adjunct Structure

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 0 | `tstart` | 8 | `real_8` | abscissa value for first record |
| 8 | `tdelta` | 8 | `real_8` | abscissa distance between records |
| 16 | `tunits` | 4 | `int_4` | units for record abscissa values |
| 20 | `components` | 4 | `int_4` | number of columns per record (max of 14) |

... continued on next page

Table 16: Type 5000 Adjunct Structure (... continued)

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 24 | t2start | 8 | real_8 | abscissa value for first column record |
| 32 | t2delta | 8 | real_8 | abscissa distance between column records |
| 40 | t2units | 4 | int_4 | units for column abscissa values |
| 44 | record_length | 4 | int_4 | length of record in bytes |
| 48 | comp | 112 | COMPSTRUCT[14] | record column definitions |
| 160 | quadwords | 96 | real_8[12] | systems modeling frame info |

**3.1.2.5.1.1 tstart** The tstart field in a Type 5000 file is interpreted in the same way as the xstart field in a Type 1000 file, except that it refers to the start time of the first record. See section 3.1.2.1.2. When the time associated with a record is stored within the record itself, this field is generally set to 0.

**3.1.2.5.1.2 tdelta** The tdelta field in a Type 5000 file is interpreted in the same way as the xdelta field in a Type 1000 file, except that it defines the interval between consecutive records in the file. See section 3.1.2.1.3.

**3.1.2.5.1.3 tunits** The tunits field in a Type 5000 file is interpreted in the same way as the xunits field in a Type 1000 file, except that it specifies units for the tstart and tdelta fields. See section 3.1.2.1.4.

**3.1.2.5.1.4 components** The components field indicates the number of columns or fields in each record within the file. Storage space limits this to a maximum of 14.

**3.1.2.5.1.5 t2start** The t2start field exists only for structural compatibility with other adjunct structures, and is not used in Type 5000 files. The field is normally set to zero.

**3.1.2.5.1.6 t2delta** The t2delta field exists only for structural compatibility with other adjunct structures, and is not used in Type 5000 files. The field is normally set to zero.

**3.1.2.5.1.7 t2units** The t2units field exists only for structural compatibility with other adjunct structures, and is not used in Type 5000 files. The field is normally set to zero.

**3.1.2.5.1.8 `record_length`** The `record_length` field defines the length of each record in bytes. Unlike Type 3000 files this must be set to the sum of the bytes occupied by each component.

**3.1.2.5.1.9 `comp`** The `comp` field contains a sequence of record column or field definitions, each of which is an eight-byte structure as defined in section 3.1.2.5.13.

**3.1.2.5.1.10 `quadwords`** The quadwords define system modeling parameters for the file as a whole. Each of the quadwords is an 8-byte value; the use of the term *quadword* is a legacy from contemporary host computers that used a native 16-bit word. Historically, different sub-types of Type 5000 stored different information in the quadwords fields. Although declared as a set of `real_8` values, in fact elements in quadwords may have varying types, depending on the file subtype. As a general rule, the first quadword should be interpreted as a `char[8]` string identifying a frame of reference.

The only subtypes explicitly supported in current Midas implementations are the ones for Type 5001 state vector files and Type 5010 geodetic reference positions files, described in section 3.1.2.5.18. There is historical evidence of a Type 5002 *frame of reference* file, but this format is not currently supported in Midas.

**3.1.2.5.2 Extended Type 5000 Files** Extended Type 5000 files are used operationally, though their use is not encouraged. These files are similar to a standard Type 5000 file, except that they are not limited to 14 `components`. Increasing the number of `components` is done by extending the size of the adjunct header beyond 256 bytes, as shown below. This extension of the adjunct header necessitates the use of a `data_start` that is greater than 512.

Table 17: Extended Type 5000 Adjunct Structure

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 0 | `tstart` | 8 | `real_8` | *no change* |
| 8 | `tdelta` | 8 | `real_8` | *no change* |
| 16 | `tunits` | 4 | `int_4` | *no change* |
| 20 | `components` | 4 | `int_4` | number of columns per record (no max limit) |
| 24 | `t2start` | 8 | `real_8` | *no change* |
| 32 | `t2delta` | 8 | `real_8` | *no change* |
| 40 | `t2units` | 4 | `int_4` | *no change* |
| 44 | `record_length` | 4 | `int_4` | *no change* |
| 48 | `comp` | 112 | `COMPSTRUCT[14]` | First 14 record column definitions |
| 160 | `quadwords` | 96 | `real_8[12]` | *no change* |
| 256 | `comp_ext` | N*8 | `COMPSTRUCT[N]` | Additional N column definitions (where N = MAX(0, `components`-14)) |

... continued on next page

Table 17: Extended Type 5000 Adjunct Structure (... continued)

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|

Although supported by NeXtMidas for over a decade Extended Type 5000 files have limited support in the other Midas baselines.

**3.1.2.5.3 `COMPSTRUCT` Structure** Each component is a sequence of column or field values. The columns are defined using the structure in Table 18.

Table 18: Type 5000 COMPSTRUCT Structure

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 0 | `name` | 4 | `char[4]` | column name |
| 4 | `format` | 2 | `char[2]` | format for column data |
| 6 | `type` | 1 | `int_1` | type of column data |
| 7 | `units` | 1 | `int_1` | column units |

**3.1.2.5.3.1 `name`** The `name` field associates a title or name with the corresponding column or field of the record. The value should be padded with spaces to four characters.

Although `name` may include any ASCII character, use of lower-case letters and non-printing characters (other than spaces to pad out names shorter than four characters) are strongly discouraged as there is limited support for them in the current Midas baselines.

**3.1.2.5.3.2 `format`** The `format` field identifies the format of data within the column; see section 3.1.1.12.1.

**3.1.2.5.3.3 `type`** The `type` field specifies the type of measurement represented by the component, using values from Table 19. This is frequently used to indicate the coordinate system for positional fields. The most common types seen are 0 (NONE) and 6 (GEODETIC). Components for which type is not applicable will use 0 (NONE).

Table 19: Type 5000 COMPSTRUCT Column Type Codes

| Code | Name | Description |
|---|---|---|
| 0 | NONE | Type is none or undefined |
| 1 | SCALAR | Value is a scalar quantity |
| 2 | CARTESIAN | Value in the Cartesian coordinate system (x, y, z) |
| 3 | SPHERICAL | Value in the spherical coordinate system (R, theta, phi) |
| 4 | CYLINDRIC | Value in the cylindrical coordinate system (R, theta, h) |
| 5 | ELLIPSOID | Value in the ellipsoidal coordinate system |
| 6 | GEODETIC | Value in the geodetic coordinate system (alt, lat, lon) |
| 7-9 | undefined | not used at this time |
| 10 | MATRIX | Value is a matrix |

**3.1.2.5.3.4 `units`** The `units` field specifies the units of the component. This is interpreted in the same way as the `xunits` field in a Type 1000 file, except that it specifies units for the individual component. See section 3.1.2.1.4. Note that space restrictions on this field limit the legal range of unit codes to no more than 127.

When the `type` is `GEODETIC`, the `units` should be set to indicate the units for the altitude component (the latitude and longitude components are always measured in degrees).

**3.1.2.5.4 Type 5001/5010 Quadwords** As noted in section 3.1.2.5.11, various Type 5000 files may store different information in their quadwords fields. The usage described in this section corresponds to what was originally a Type 5001 *object data* file, but is now used more commonly to encode state vectors in various frames of reference.

Table 20: Type 5001/5010 Quadword Adjunct Structure

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 0 | `padding` | 160 | `char[160]` | Fill overlaying type 5000 adjunct header prefix |
| 160 | `frame_of_ref` | 8 | `char[8]` | coordinate reference frame |
| 168 | `altitude` | 8 | `real_8` | altitude (meters) for topocentric |
| 176 | `latitude` | 8 | `real_8` | latitude (deg) for topocentric |
| 184 | `longitude` | 8 | `real_8` | longitude(deg) for topocentric |
| 192 | `azimuth` | 8 | `real_8` | azimuth(deg) for topocentric |
| 200 | `elevation` | 8 | `real_8` | elevation (deg) for topocentric |
| 208 | `roll` | 8 | `real_8` | roll (deg) for topocentric |

... continued on next page

Table 20: Type 5001/5010 Quadword Adjunct Structure (... continued)

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 216 | *not used 1* | 8 | real_8 | unused |
| 224 | *not used 2* | 8 | real_8 | unused |
| 232 | epoch_year | 8 | real_8 | year of epoch for ECI |
| 240 | epoch_seconds | 8 | real_8 | seconds of epoch for ECI |
| 248 | hour_angle | 8 | real_8 | GW angle (rad) at epoch for ECI |

**3.1.2.5.4.1 frame_of_ref** The frame_of_ref field specifies the frame of reference used for the file. Standard reference frames are listed in Table 21.

Table 21: Type 5001/5010 Quadword frame_of_ref Values

| Name | Description |
|---|---|
| **ECR** | Earth Centered Rotating. X and Y in equatorial plane at 0 and 90 degrees east respectively, and Z north along rotational axis to complete right-hand system |
| **ECI** | Earth Centered Inertial. X towards vernal equinox, Z north along rotational axis, Y completes right-hand system (90 degrees east of X) |
| **TOPOCENT** | Topocentric. For Cartesian positions, use X due south in local horizon plane, Y east in horizon plane, and Z perpendicular (up) to local horizon to complete right-hand system |
| **TOP** | Alternative topocentric. For Cartesian positions, use X due East in local horizon, Y north in horizon plane, and Z perpendicular (up) to local horizon. This system is then augmented by a specified azimuth rotation, elevation angle, and roll angle |

The frame of reference may be further refined by the type field of the COMPSTRUCT record, for example, to indicate whether the value is presented in Cartesian, spherical, cylindrical, geocentric, or geodetic form.

If the reference name does not match one of these recognized names, it is assumed to be a custom frame of reference. This document does not attempt to specify how such a file should be interpreted, though it is reasonable to assume that the values stored in the quadwords field may not be consistent with the structure described in Table 20. A special Type 5000 file named by the frame_of_ref value may be available to provide transformation matrices used in the interpretation of the file contents.

A few systems have been known to generate Type 5010 files with a POS column that has a type of GEODETIC and a frame_of_ref left blank. Although such is not strictly legal, it is however unambiguous since GEODETIC overrides frame_of_ref. If such a file is encountered, processing systems are encour-

aged to handle it as if `frame_of_ref` was set to `ECR`.

**3.1.2.5.4.2 `altitude`** The `altitude` field specifies the altitude above the reference ellipsoid of the origin of a topocentric frame of reference, in meters. It applies to the **`TOPOCENT`** and **`TOP`** reference frames.

**3.1.2.5.4.3 `latitude`** The `latitude` field specifies the geodetic latitude of the origin of a topocentric frame of reference, in degrees. It applies to the **`TOPOCENT`** and **`TOP`** reference frames.

**3.1.2.5.4.4 `longitude`** The `longitude` field specifies the geodetic longitude of the origin of a topocentric frame of reference, in degrees. It applies to the **`TOPOCENT`** and **`TOP`** reference frames.

**3.1.2.5.4.5 `azimuth`** The `azimuth` field specifies the reference azimuth in degrees. It applies to the **`TOP`** reference frame.

**3.1.2.5.4.6 `elevation`** The `elevation` field specifies the reference elevation angle in degrees. It applies to the **`TOP`** reference frame.

**3.1.2.5.4.7 `roll`** The `roll` field specifies the reference roll angle in degrees. It applies to the **`TOP`** reference frame.

**3.1.2.5.4.8 `epoch_year`** The `epoch_year` field specifies the epoch year for frames of reference tied to the vernal equinox. It applies to the **`ECI`** reference frame.

**3.1.2.5.4.9 `epoch_seconds`** The `epoch_seconds` field specifies the epoch time since the beginning of the epoch year (excepting leap-seconds which are not counted) for frames of reference tied to the vernal equinox. It applies to the **`ECI`** reference frame.

Although specified as a number of seconds since the beginning of the epoch year, the value must correspond to an integral number of days (i.e., the value must be evenly divisible by 86,400). Current versions of X-Midas and NeXtMidas will encounter issues when reading/writing `epoch_seconds` values that are not a multiple of 86,400.

**3.1.2.5.4.10 `hour_angle`** The `hour_angle` field specifies the Greenwich sidereal time of epoch or equivalently, the hour angle of the vernal equinox at epoch as viewed from Greenwich, in radians. It applies to the **`ECI`** reference frame.

Note that the hour_angle is directly related to the epoch_year and epoch_seconds by the following algorithm:

```
// Computes hour_angle given the number of UTC days since
// 1 Jan 1950 based the a NORAD algorithm from Dec 1986.
RAGW (DS50) {
  CR     = 1.72027915249490159E-2
  THGR70 =    1.73213060363866E0
  TPI    = 6.28318530717958648E0
  FK4R   =    5.06494548928754E-15

  TS70   = DS50 - 7304
  DS70   = floor(TS70)
  RAGW   = THGR70 + CR*DS70 + (CR+TPI)*(TS70-DS70) + TS70*TS70*FK4R

  // This next line uses the remainder function as prescribed by the
  // IEEE 754 standard (known as "Math.IEEEremainder(..)" in Java or
  // "DMOD(..)" in Fortran)
  RAGW = remainder(RAGW,TPI)

  if (RAGW<-.5*TPI) RAGW += TPI
  if (RAGW>+.5*TPI) RAGW -= TPI

  return RAGW
}
```

It is an error to use an hour_angle that is inconsistent with the equation above since some systems only consider hour_angle (ignoring epoch_year and epoch_seconds), while other systems consider only epoch_year and epoch_seconds (computing the hour angle as required).

**3.1.2.5.5 Categories of Type 5000 Files** Since Type 5000 files are frequently used to plot geodetic positions on a map there are a few special categories of Type 5000 files that have special handling by XGEOPLOT in X-Midas and PLOT in NeXtMidas. These categories are identified based on the set of components defined in the adjunct header field. Table 22 characterizes the categories based on component names. In addition to the components listed, the **NOTE**, **CONN**, and **HDG** components may be present as well; the NeXtMidas PLOT primitive will use these components, while XGEOPLOT ignores them. Users are strongly encouraged to follow these conventions when creating Type 5000 files that contain data in these categories so that these Midas primitives can be used to display the data effectively.

Table 22: Common Type 5000 File Categories

| Category | Standard Components |
|---|---|
| Position | **NAME**, **POS** |
| Ellipse | **NAME**, **POS**, **ELPS** |
| Ellipse Flags | **NAME**, **POS**, **ELPS**, **FLAG** |
| Cone | **NAME**, **POS**, **POSP**, **CONE** |
| Cone Flags | **NAME**, **POS**, **POSP**, **CONE**, **FLAG** |
| State Vector | **POS**, **VEL** [, **ACC**] [, **TIME**] |

A number of applications that read in Type 5001 State Vector files assume that **POS**, **VEL**, and **ACC** are present, appear in that order, and have format `VD`. Accordingly this is acknowledged as a "*de facto* preferred usage" for Type 5001 files; however it is neither universal nor mandatory, and all readers of Type 5001 files are encouraged to verify the file structure (as described in the HCB) prior to processing.

The standard components referenced in Table 22 are defined in Table 23.

Table 23: Component Definitions for Common Type 5000 File Categories

| Component Name | Format | Description |
|---|---|---|
| **NAME** | **1A** | Name of the point/cone/ellipse |
| **NOTE** | **1A** | Additional notes about the point |
| **POS** | **VD** | Position (altitude, latitude, longitude) if using `GEODETIC` or (x,y,z) if using *CARTESIAN* |
| **VEL** | **VD** | Velocity (x,y,z) if using *CARTESIAN* |
| **ACC** | **VD** | Acceleration (x,y,z) if using *CARTESIAN* |
| **POSP** | **VD** | Secondary cone position (altitude, latitude, longitude) |
| **ELPS** | **VD** | Ellipse parameters (azimuth, elevation, roll) |
| **CONN** | **SB** | If the value is one, a line will be drawn connecting this point to the previous point when plotting. If the value is zero, no line will be drawn. Behavior for other values is undefined |
| **HDG** | **SF** | Cardinal heading to use for the symbol (0 to 360) |
| **FLAG** | **8B** | Plotting flags |
| **TIME** | **SD** | Time of state-vector point (seconds since 1 January 1950); if omitted, time of state-vector point can be computed based on the file start time and `xdelta` |

The **NAME** and **NOTE** components can be ASCII fields of any length. **1A** and **2A** are the most common

formats for **NAME** components, while **NOTE** components are usually longer.

The **POS** and **POSP** components list latitude and longitude in degrees, but the altitude may be in meters or feet (indicated by the `units` field for the component).

**CONN** and **HDG** are used to represent flight paths. For a sequence of positions representing a flight path, the **CONN** field would be set to 1 for all but the first point. The **HDG** field indicates the direction in decimal degrees, clockwise from north, and may be used to display a directional symbol.

The **FLAG** component stores an array of eight *int_1* values that are used to customize the representation of each record on the plot. The purpose of each element is listed in Table 24; for the semantics of each element see the XGEOPLOT (X-Midas) or PLOT (NeXtMidas) explain pages.

Table 24: Type 5000 Common FLAG Component Structure

| Flag Index | Description |
|---|---|
| 0 | Length of the name (0 through 9) |
| 1 | Name position |
| 2 | Foreground color (by index number) |
| 3 | Background color |
| 4 | Cone/ellipse representation |
| 5 | Invert |
| 6 | Representation symbol |
| 7 | Symbol size |

**3.1.2.6 Type 6000** Type 6000 files contain a series of non-homogeneous records, all with the same structure, much like Type 3000 files except with fewer restrictions. Specifically, Type 6000 files permit more columns per record, allow a longer column name, and permit columns that store fixed-length arrays of data as well as single data elements.

**3.1.2.6.1 Adjunct Structure** Type 6000 files use the Type 3000 file adjunct structure, though some fields are interpreted differently or remain undefined. The `subr` field for a Type 6000 file should not be relied on, but systems creating Type 6000 files should attempt to set it to represent the subrecord structures as accurately as possible within the limitations of the Type 3000 format. Instead, the structure of the Type 6000 record is defined in the **SUBREC_DEF** keyword, as described in section 3.1.2.6.12, with the structure of the **SUBREC_DEF** value itself determined by the value of the **SUBREC_DESCRIP** keyword as described in section 3.1.2.6.13.

**3.1.2.6.2 TYPE0 Subrecord Definition Layout** In a **TYPE0** subrecord definition, each record field is defined using a ninety-six byte text block which defines the name, location in the record, format, and

length of the field. In Midas implementations, the record description is extracted from the extended header and converted into a binary format for internal use. The binary format does not appear within any BLUE file; however, the act of converting to binary format imposes some limitations on the values that can be represented in the textual column description. The layout of the definition including the types used for internal representation in X-Midas is provided in Table 25. Each field must be padded to its full length with spaces (for string values) or leading zeros (for numeric values)

The column definitions in a **`TYPE0`** subrecord keyword value field need not be in strictly increasing offset order.

Table 25: Type 6000 TYPE0 Subrecord Format

| Offset | Name | Size | Type | Description |
|---|---|---|---|---|
| 0 | `name` | 24 | `char[24]` | column/field name |
| 24 | `minval` | 24 | `char[24]` | minimum value in column |
| 48 | `maxval` | 24 | `char[24]` | maximum value in column |
| 72 | `offset` | 8 | `char[8]` | byte offset into record |
| 80 | `num_elts` | 4 | `char[4]` | number of elements in column |
| 84 | `units` | 4 | `char[4]` | column data units code |
| 88 | `format` | 2 | `char[2]` | column format code |
| 90 | `uprefix` | 3 | `char[3]` | units Greek Prefix code |
| 93 | `reserved` | 3 | `char[3]` | pad to 96-byte total length |

**3.1.2.6.2.1 `name`** The `name` field contains a name for the field or column, which must be padded to the full 24-byte length with spaces.

Although `name` may include any ASCII character, use of lower-case letters and non-printing characters (other than spaces to pad out names shorter than 24 characters) are strongly discouraged as there is limited support for them in the current Midas baselines.

**3.1.2.6.2.2 `minval`** The `minval` field can be used to store the minimum value of the set of data values stored in the column in this file. This value must be computed by scanning the entire file, and as such the value is routinely zero or not valid. The field can be either zero or all space when no value is known. The contents of this field may not be properly updated or reset by systems that modify the contents of Type 6000 files, so the field value should not be trusted. The value is a `real_8` represented in text.

**3.1.2.6.2.3 `maxval`** The `maxval` field can be used to store the maximum value of the set of data values stored in the column in this file. This value must be computed by scanning the entire file, and as such the value is routinely zero or not valid. The field can be either zero or space when no value is known. The

contents of this field may not be properly updated or reset by systems that modify the contents of Type 6000 files, so the field value should not be trusted. The value is a `real_8` represented in text.

**3.1.2.6.2.4 `offset`** The `offset` field specifies the offset of the start of the column relative to the beginning of the record. The value is a positive integer represented in text.

**3.1.2.6.2.5 `num_elts`** The `num_elts` field permits a column to hold a fixed-length array of values, rather than a single value. The value is a non-negative integer represented in text.

**3.1.2.6.2.6 `units`** The `units` field specifies the units associated with data values in the column. The value is a three-digit integer representing an entry in Table 29.

**3.1.2.6.2.7 `uprefix`** The `uprefix` field was placed in the header to allow for multipliers to be applied to each data item. Use of this field is discouraged as all data written to files should be in natural units, and there is no documentation that explains how this value should be interpreted. The `uprefix` field should be set to **`000`**.

**3.1.2.6.2.8 `format`** The `format` field specifies the format of column elements, as described in section 3.1.1.12.1.

#### 3.1.2.6.3 Keywords

**3.1.2.6.3.1 SUBREC_DEF** This keyword has an ASCII array value which consists of subrecords consecutive subrecord definition structures, one for each record field. The layout and interpretation of the subrecord definition structure is specified by the **`SUBREC_DESCRIP`**keyword.

**3.1.2.6.3.2 SUBREC_DESCRIP** For the purposes of generality, MARTES introduced the use of the keyword **SUBREC_DESCRIP**, to allow alternative mechanisms to describe record fields. The value of the **`SUBREC_DESCRIP`**keyword is a string that names a subrecord description format. The only format currently accepted as a BLUE standard is **`TYPE0`**, which is described in section 3.1.2.6.2. If a Type 6000 file does not contain a SUBREC_DESCRIP keyword, it should be assumed that the **`TYPE0`** layout is being used. MARTES systems may require the presence of the **`SUBREC_DESCRIP`**keyword. Future versions of the BLUE format may support additional subrecord layouts, which will be identified by different values for the **`SUBREC_DESCRIP`**keyword.

**3.1.2.6.4 Standard Column Names for Type 6000 Files** To allow the exchange of Type 6000 files between applications, standard field names are used. These names are not keyword names and are only reserved for Type 6000 field names. The known reserved field names are described below. Applications wishing to exchange Type 6000 files that contain data of these types are encouraged to use these standard column names.

**3.1.2.6.4.1 ADDRESS** The **ADDRESS** field contains the abscissa index where the first sample of data burst is stored in an associated Type 1000 file.

**3.1.2.6.4.2 DATA** This field provides for arbitrary data value(s). The interpretation of the **DATA** field is application dependent; the **NUM_ELTS** field specifies how many data values are present in each record.

**3.1.2.6.4.3 DTOA** The delta time-of-arrival is captured in the **DTOA** field. It is the time difference between two successive times-of-arrival (TOAs). It can also represent the time/time-of-arrival (TTOA) for current pulse computed as the time between the current TOA and the next TOA.

**3.1.2.6.4.4 FREQ** Frequency is specified using this field.

**3.1.2.6.4.5 FREQ_PRF** The pulse repetition frequency of the data is identified through this field.

**3.1.2.6.4.6 MAGN** This field is for the magnitude.

**3.1.2.6.4.7 NUMBER_SAMPLES** This field indicates the number of samples in a data burst. Samples are stored in an associated Type 1000 file.

**3.1.2.6.4.8 PA** This field provides for the pulse amplitude of the input data.

**3.1.2.6.4.9 PRI** Pulse repetition interval, i.e., the reciprocal of **FREQ_PRF**. If the engineering units specified for this component is counts, the header `rdelta` field is used to convert to time in seconds.

**3.1.2.6.4.10 PW** This field is for the pulse width (i.e., pulse duration or PD). If the engineering units specified for this component is counts, the header rdelta field is used to convert to time in seconds.

**3.1.2.6.4.11 RANGE** This field provides for the inclusion of range to target data.

**3.1.2.6.4.12 TARGET_ID** This field allows for inclusion of the target designator code/waveform mode code.

**3.1.2.6.4.13 TIME** This field is the time value in seconds which may be relative to the start of the file or to 1 January 1950. **TIME** values need not be monotonically increasing in the file.

Time values less than 3.1536E7 (one year) are generally considered to be relative to the file and those greater are considered to be relative to 1 January 1950. In the case where **TIME** is relative to the start of the file, the handling of leap-seconds is undefined.

**3.1.2.6.4.14 TOA** This provides for the Time Index, the TOA of a pulse or of a data burst. If the engineering units for this parameter are counts, the header field rdelta is used to convert time to seconds. Any TOA values in the file should be monotonically increasing. For time based data, the time index field must be the first sub-record.

**3.1.2.6.4.15 VELOCITY** This field allows for the input of velocity data.

**3.1.2.6.4.16 WSEC and FSEC** Time value in whole seconds since 1 January 1950 (**WSEC**) and fractional seconds (**FSEC**). The values for **WSEC** will be non-negative and the values for **FSEC** will be in the range [0.0,1.0). The time values conveyed need not be monotonically increasing in the file.

This format is used as an alternative to **TIME** in situations where the ability to convey times with a precision better than 1 microsecond is required.

**3.1.2.6.4.17 XTAL_DIV** This is the crystal divider. The file header should contain a keyword **XTAL_FREQ** to derive this value from **FREQ_PRF**.

**3.1.2.7 Long Column Names** Long column names is an optional way of permitting longer names than the subrecords (or components) of a Type 3000, Type 5000, and/or Type 6000 file permit. When the long column name support is used, the first 3 characters of the name (23 characters for a Type 6000 file) are stored in the normal `name` field preceded by an ASCII tilde ("~"). The full name is then stored in an extended header keyword.

The long names are stored in the extended header in keywords named `SRn` (where n is the one-based subrecord number) that are within the `SECTION=SUBRECORD_NAMES` ... `SECTION=END` section.

For example, a Type 3000 with subrecords named “FRED”, “WILMA”, “DINO”, and “PEBBLES” would have the `name` fields in the HCB listed as “FRED”, “~WIL”, “DINO”, and “~PEB” and the Extended

Header Keywords:

```
SECTION=SUBRECORD_NAMES
SR2=WILMA
SR4=PEBBLES
SECTION=END
```

Readers of BLUE 1.1 files that do not support long column names should use shortened forms stored in the HCB, though this may reduce clarity and may cause column names to be duplicated where two or more start with the same series of letters.

## 3.2 Data Block

The data block contains the file data in the format described by the HCB format and adjunct fields; see section 3.1.2.

## 3.3 Extended Header

The extended header provides a way to associate arbitrary metadata with the file as a sequence of keywords, or tag/value pairs. X-Midas and MARTES used different representations for keywords, although both systems should now recognize and process both types. For newly-created BLUE files, the X-Midas binary format is preferred.

See also section 3.1.1.24 for the format of keywords stored within the header control block.

### 3.3.1 X-Midas Binary Keywords

The X-Midas binary keyword format consists of a keyword header followed by the tag and the value as shown in Table 26.

Table 26: Binary (X-Midas) Keyword Structure

| Offset | Name | Length | Internal Type | Description |
|---|---|---|---|---|
| 0 | `lkey` | 4 | `int_4` | total length of keyword |
| 4 | `lext` | 2 | `int_2` | length of non-value part of keyword |
| 6 | `ltag` | 1 | `int_1` | length of keyword tag |
| 7 | `type` | 1 | `char` | type of data in keyword value |
| 8 | `value` | `lkey - lext` | *depends on type* | keyword value |
| 8 + `lkey - lext` | `tag` | `ltag` | `char[]` | keyword tag |
| `lkey` - *rem* | *padding* | *rem* | `int_1` | padding to reach multiple of eight bytes |

Padding is required to reach a total keyword structure size that is a multiple of eight bytes; the value *rem* is the non-negative remainder of (8 + `lkey` - `lext` + `ltag`) divided by eight. The keyword is required to start on a 4-byte boundary, but should generally be on an 8-byte boundary.

The keywords are read by using the keyword length as a method of incrementing to the next keyword. This ensures that any keyword using a type not recognized by a BLUE file processor can at least be skipped over or passed through uninterpreted.

**3.3.1.1 `lkey`** The `lkey` field contains the total length of this keyword, including the header and any padding added to meet alignment requirements.

**3.3.1.2 `lext`** The `lext` field indicates any space that is not part of the data: the eight byte header, the keyword tag, and any padding required to make the keyword length a multiple of eight bytes. Thus the length of the data, which follows the keyword, is equal to the keyword length minus the extra length.

**3.3.1.3 `type`** The `type` field provides the data format type for individual elements in the keyword value. The data format follows the standard MIDAS type formats described in Table 6. Not all formats are supported: in particular, keywords cannot have offset byte values. The type designator **A** indicates a string with length determined by the size of the value region; the eight-character implication of the type designator does not apply in keywords. No size designator is associated with the data format; size is implicitly determined by the type size and value length.

**3.3.1.4 `value`** The `value` region contains the value that is associated with the tag by this keyword. In the case of a keyword with an array value, the number of elements is calculated by dividing the size of the data block by the element block size. Values are represented in the header byte order, as specified by the `head_rep` field (see section 3.1.1.2).

Although **A** permits use of any ASCII characters, the use of non-printing characters (other than the ASCII space) is strongly discouraged as there is inconsistent support for them in the Midas baselines. The Platinum standard gives special meanings to ASCII unit separator (US) character (0x1F) when used within this context; implementations of this standard may optionally support the Platinum usage of the ASCII US character.

**3.3.1.5 `tag`** The `tag` region provides the keyword's tag, or key name. The tag immediately follows the value, without intervening padding. Padding required to make the total keyword length a multiple of eight bytes follows the tag.

Although `name` may include any ASCII character, use of lower-case letters and non-printing characters are strongly discouraged as there is limited support for them in the current Midas baselines.

### 3.3.2 MARTES ASCII Keywords

MARTES ASCII Keywords are not supported within BLUE 1.1 files. Files produced with MARTES ASCII Keywords were phased out between 2002 and 2005; subsequent to that, all BLUE files produced by MARTES have used the Binary Keyword format. Please see the BLUE 1.0 standard for more details.

## 3.4 Standard BLUE Keywords

Keywords in this section are optional, but if they are to have their described meaning, they must be present in the HCB keyword area and encoded in the ASCII format described in section 3.1.1.24. Because space in the HCB keywords area is limited, only those keywords explicitly listed below belong in the HCB keywords area; all other keywords belong in the Extended Header (see section 3.3).

All applications are strongly encouraged to use the **IO** and **VER** keywords. The current versions of X-Midas and NeXtMidas will automatically include **IO** and **VER**, but have options to permit their omission where necessary for backwards compatibility. If the HCB keywords block is full, these frameworks will delete (or refuse to insert) user-defined keywords in the HCB in order to make room for them. In cases where a keyword is deleted, X-Midas and NeXtMidas will typically issue a warning.

If **IO** or **VER** are not found when reading a file, `IO=UNKNOWN` or `VER=1.0` are to be assumed.

### 3.4.1 Main Header Keywords

The keywords in this section are optional, but if they are to have their described meaning must be present in the HCB keyword area, encoded in the ASCII format described in section 3.1.1.24.1. Because space in the HCB keyword are is limited, systems that generate BLUE files are encouraged to place any keywords not described in this section into the extended header.

**3.4.1.1 CREATOR** The **CREATOR** keyword is used by NeXtMidas to indicate the Midas implementation that last updated the file. The value of the keyword is three ASCII characters representing the Midas system (**NXM** for NeXtMidas) followed immediately by an ASCII representation of the version of that system (e.g., **2.6.0**). The three characters may be a digraph (e.g., **XM**), in which case the third character should be an ASCII space.

As a general rule, presence of this keyword indicates that the file was last updated by a NeXtMidas installation, but NeXtMidas will omit the **CREATOR** keyword if there is insufficient room to add it in the main header. X-Midas does not currently use this keyword, and will not remove it from the header if it subsequently modifies a file that contains it.

Although the **CREATOR** keyword has a similar function to the Platinum **IOVERSION**, the **CREATOR** has a fixed-form value which enables programmatic version checking (e.g., to work around known bugs in specific releases, etc.) whereas **IOVERSION** is free-form.

The **CREATOR** keyword has been supported by NeXtMidas, in its current from, since release 2.1.0 (2005). Prior versions of NeXtMidas included the **CREATOR** keyword, but omitted the dots in the version number

(e.g., `NXM180` rather than `NXM1.8.0`) -- this legacy syntax is not supported in BLUE 1.1 files.

**3.4.1.2 IO** The **IO** keyword was introduced in the Platinum extensions to BLUE, and serves to represent the implementation or library that created or modified the file. It may appear in Midas BLUE 1.0 files, and will appear in BLUE 1.1 files (space permitting), but should be taken as informational only. Midas systems may or may not remove or update this keyword if they modify a BLUE file that contains it.

For consistency, the current Midas baselines use the following values (note that this chart is for informational purposes only as these values are neither exclusive nor mandatory):

Table 27: IO Values for Midas Baselines

| **IO Value** | **Midas Baseline** |
|---|---|
| `IO=NeXtMidas` | NeXtMidas |
| `IO=X-Midas` | X-Midas |

Note that the BLUE 1.0 standard does not require the use of the **IO** keyword, and as a result nearly all files conforming to that standard omit it.

**3.4.1.3 PACKET** The **PACKET** keyword indicates that a separate file contains packetized data that controls the interpretation of data in this file. The keyword value identifies a packet handler and any switches that modify its behavior. Packet handlers are an extension supported by NeXtMidas. Common packet handler identifiers are **ICE** and **SDDS**. For a list of supported packet handlers, their switches, and how they affect data interpretation please see the NeXtMidas documentation.

**3.4.1.4 PKT_BYTE_COUNT** This keyword is by X-Midas to indicate that a file contains packetized data. The value of the keyword is the total number of bytes of packetized data contained in the file.

**3.4.1.5 TC_PREC** Precision time with more than microsecond resolution is provided by the **TC_PREC** keyword in the main header keyword block. The keyword value is a string containing a decimal textual representation of a 64-bit floating point number that provides an adjustment in fractional seconds to be added to the epoch specified in the HCB `timecode` field. The absolute value of this adjustment should be less than $10^{-6}$ since the `timecode` field already contains a value precise to the microsecond. A negative value is permitted. The addition of this value provides an epoch time accurate to picoseconds. The value should not be used to provide times finer than picoseconds, since some Midas processors truncate the additional precision under the assumption that it results from floating point representation errors. In the Midas infrastructure, library routines that extract precision time from the header consult both the `timecode` field and **TC_PREC** keyword, and return time as two 64-bit floating point numbers, integral and fractional seconds, so that accuracy to picoseconds is not lost.

In situations where the timecode field is sufficient to accurately describe the start time of the file, the **TC_PREC** keyword may either be omitted or may be set to 0.0.

**3.4.1.6 VER** The **VER** keyword was introduced in the Platinum extensions to BLUE and serves to indicate conformance of the file to a particular version of the Platinum specification. It may appear in Midas BLUE 1.0 files and will appear in BLUE 1.1 files (space permitting). Midas systems may or may not remove or update this keyword if they modify a BLUE file that contains it.

Table 28: VER Values for Current Versions of BLUE and Platinum

| VER Value | Specification |
|---|---|
| VER=1.0 | BLUE 1.0 |
| VER=1.1 | BLUE 1.1 *(this document)* |
| VER=2.0 | Platinum |

Note that the BLUE 1.0 standard does not require the use of the **VER** keyword, and as a result, nearly all files conforming to that standard omit it.

### 3.4.2 Extended Header Keywords

The keywords in this section generally appear in the extended header.

**3.4.2.1 ACQDATE** The **ACQDATE** keyword represents the date of the acquisition as a six character string of the format YY.DDD where YY is the two-digit year and DDD is the day number in the year where 1 January is 001. If the value of YY is less than 50, the century adjustment is assumed to be 2000; otherwise, the century adjustment is assumed to be 1900. Leading zeroes are required so that the date is exactly six characters long.

The keyword corresponds to the legacy GOLD format acqdate header field. The effect of the presence of this keyword is determined by individual processing systems. Some processing systems expect this keyword to be in the main header.

MARTES permitted **ACQDATE** to use YYMMDD and YYYYMMDD (with leading zeros as appropriate) as an alternate to the YY.DDD syntax. Consequently some of the widely-used Platinum libraries are known to insert an **ACQDATE** using the YYYYMMDD form. Accordingly, BLUE 1.1 implementations may permit YYYYMMDD as an alternate syntax where necessary for compatibility with Platinum-based systems.

**3.4.2.2 ACQTIME** The **ACQTIME** keyword represents the time of the acquisition as an eight-character string of the format HH:MM:SS, representing hours, minutes, and seconds. Leading zeros are required so that the time is exactly eight characters long. The keyword corresponds to the legacy GOLD format

`acqtime` header field. The effect of the presence of this keyword is determined by individual processing systems. Some processing systems expect this keyword to be in the main header.

**3.4.2.3 COMMENT** The **COMMENT** keyword provides a free-form comment that accompanies the file. Some systems (including NeXtMidas and X-Midas) will display the value of the **COMMENT** when listing details about the file, even in situations where keywords would not normally be listed.

Some processing systems expect this keyword to be in the Main Header; however, its inclusion in the Main Header is to be on a strictly space-available basis (i.e., the inclusion of the **COMMENT** in the Main Header should never be at the expense of one of the keywords that is required to be in the Main Header).

**3.4.2.4 SUBREC_DEF** **SUBREC_DEF** is used in Type 6000 files to define the structure of individual records. See section 3.1.2.6.12.

**3.4.2.5 SUBREC_DESCRIP** **SUBREC_DESCRIP** is used in Type 6000 files to define the structure of a **SUBREC_DEF** keyword value. See section 3.1.2.6.13.

**3.4.2.6 T4INDEX** **T4INDEX** is used in Type 4000 files to locate the starts of individual keyword blocks. See section 3.1.2.4.15.

**3.4.2.7 TIMELINE** The **TIMELINE** keyword is used by NeXtMidas to store timecode information when the samples are not linear in time.

## 3.5 Other Constants

### 3.5.1 Unit codes

BLUE files use an integer code to specify the units for various measurements. These codes are listed in Table 29. Italicized codes use non-SI units for which there is a common SI equivalent, and should be avoided where possible. The maximum supportable code value is 127, since these values are stored in an `int_1` field within a Type 5000 header. Note that some codes also distinguish different applications of the units, e.g., 34, 60, and 61 all denote degrees, but the latter two also indicate that the value relates to a geographic location.

Table 29: Midas Codes to Identify Value Units

| Code | Units | Name |
|---|---|---|
| 0 | | Not applicable |
| 1 | s | Time (seconds) |
| 2 | s | Delay (seconds) *(less common then 1 (Time) since "delay" is usually implied by context)* |
| 3 | Hz | Frequency (hertz) |
| 4 | tcode | Time code format |
| 5 | m | Distance (meters) |
| 6 | m/s | Speed (meters per second) |
| 7 | m/s^2 | Acceleration (m/s per second) |
| 8 | m/s^3 | Jerk (m/s^2 per second) |
| 9 | Hz | Doppler (hertz) |
| 10 | Hz/s | Doppler rate (hertz per second) |
| 11 | J | Energy (joules) |
| 12 | W | Power (watts) |
| 13 | g | Mass (grams) |
| 14 | dm^3 | Volume (dm^3 = cubic decimeter = liter) |
| 15 | W/sr | Angular power density (watts per steradian) |
| 16 | W/rad | Integrated power density (watts per radian) |
| 17 | W/m^2 | Spatial power density (watts per square meter) |
| 18 | W/m | Integrated power density (watts per meter) |
| 19 | W/MHz | Spectral power density (watts per megahertz) |
| 30 | unk | Unknown |
| 31 | none | General dimensionless |
| 32 | counts | Counts |
| 33 | rad | Angle (radians) |
| 34 | deg | Angle (degrees) |
| 35 | dB | Relative power (decibels) |
| 36 | dBm | Relative power (decibels relative to 1 milliwatt) |
| 37 | dBW | Relative power (decibels relative to 1 watt) |
| 38 | sr | Solid angle (steradian) |
| *40* | *ft* | *Distance (US foot = 25.4mm)* |
| *41* | *nmi* | *Distance (nautical mile = 1852m)* |
| *42* | *ft/s* | *Speed (feet per second)* |
| *43* | *nmi/s* | *Speed (nautical miles per second)* |
| *44* | *kt* | *Speed (knots = nautical miles per hour)* |

... continued on next page

Table 29: Midas Codes to Identify Value Units (... continued)

| Code | Units | Name |
|---|---|---|
| *45* | *ft/s^2* | *Acceleration (feet per second per second)* |
| *46* | *nmi/s^2* | *Acceleration (nautical mile per second per second)* |
| *47* | *kt/s* | *Acceleration (knots per second)* |
| 48 | g | Acceleration (gravities = 9.8 m/s^2) |
| 49 | g/s | Jerk (gravities per second) |
| 50 | rps | Rotation or revolution rate (rotations per second) |
| 51 | rpm | Rotation or revolution rate (rotations per minute) |
| 52 | rad/s | Angular velocity (radians per second) |
| *53* | *deg/s* | *Angular velocity (degrees per second)* |
| 54 | rad/s^2 | Angular acceleration (radians per second per second) |
| *55* | *deg/s^2* | *Angular acceleration (degrees per second per second)* |
| 56 | % | Percentage |
| 57 | psi | Pressure (pounds per square inch = 6.894757 kPa) |
| *58* | | *unknown (reserved for future use)* |
| *59* | | *unknown (reserved for future use)* |
| 60 | deg | Latitude (in a range of [-90,+90] where +90 is the North Pole and -90 is South Pole) |
| 61 | deg | Longitude (in a range of [-180,+180] where negative values measure west of the Prime Meridian and positive values measure east of the Prime Meridian; note that -180.0 and +180.0 are both valid descriptions of the 180th Meridian) |
| *62* | *ft* | *Altitude (feet)* |
| 63 | m | Altitude (meters) |
| 64 | unk | Unknown |
| 65 | unk | Unknown |
| 66 | sym/s | Baud (symbols per second) |
| 67 | bits/s | Bit rate (bits per second) |
| 68 | V | Potential difference (volts) |
| 69 | A | Current (amps) |
| 70 | ohms | Resistance (ohms) |
| 71 | F | Capacitance (farads) |
| 72 | H | Inductance (henries) |
| 73 | K | Temperature (kelvin) |
| 74 | Pa | Pressure (pascal) |
| 75 | dBmi | Relative power (dBm referenced to isotropic source) |

... continued on next page

Table 29: Midas Codes to Identify Value Units (... continued)

| Code | Units | Name |
|---|---|---|
| 76 | dBmi/Hz | Relative power density (dBm referenced to isotropic source per hertz) |

It should be noted that the Platinum standard does not support unit codes `60` through `63` and gives special meaning to code `32` when used with time-valued quantities.

# 4 Document Information

| | |
|---|---|
| *Version* | 1.1.1 |
| *Revision* | $Revision: 327 $ |
| *Release Date* | 11-May-2018 |
| *Document Owner* | Virginia Cevasco<br>Booz Allen Hamilton<br>MAINSTRIKE Program Manager<br>853-2327 |

## 4.1 Revision History

- 17-Mar-2006 (1.0.0): Original release of Version 1.0.0
- 17-Aug-2008 (1.0.1) : Converted from Microsoft Word to restructured text to simplify generation of PDF and HTML from single source. No content changes.
- 17-Aug-2008 (175): Correct the following errors:
  - assorted typographical errors
  - size of `keywords` field in Table 2
  - IEEE floating point standard number in Table 3
  - type of `padding` in Table 20
  - all fields in Table 25 are ASCII text; the table erroneously listed their decoded types. Also correct table name to reflect that it represents **`TYPE0`** subrecords

  - semantics of `xstart` and `ystart` were interchanged in section 3.1.2.2.1
  - refine the description of the Data and Extended Header blocks in section 3 to address order and spacing
  - reword legacy floating point text in section 3.1.1.2
  - note semantic extensions in section 3.1.1.4
  - add Type 5001 to Table 4
  - in section 3.1.1.12.2, mention unsupported unsigned integer types, clarify packed bit encoding, and note padding expectations for variable-length ASCII strings
  - clarify the relationship between `timecode`, `xstart`, and **TC_PREC** in section 3.1.1.14
  - provide details on the **FLAG** component in section 3.1.2.5.29
  - add main header keywords **CREATOR** (3.4.1.1), **IO** (3.4.1.2), **PACKET** (3.4.1.3), **PKT_BYTE_COUNT** (3.4.1.4), and **VER** (3.4.1.6), and extended header keyword **TIMELINE** (3.4.2.7)
  - note difference in NeXtMidas representation and processing of **SP** data in section 3.1.1.12.2
  - Internal review release as Version 1.0.1

- 12-Sep-2008 (187):
  - Uniformly use decimal values for offsets in tables
  - Further clarifications of some updates from 175.

- 10-Dec-2008 (213):
  - Update units table to use correct SI unit symbols
  - Add references to Platinum

- 20-Jan-2010 (309):
  - Note that `r2units` in Type 3000 files is present for structural reasons only
  - Make HTML formatting prettier
  - Release as Version 1.0.2

- 18-Nov-2013 (316):
  - Updated info for file Types 1200 and 2200
  - Various keyword information added, specifically for **IO**, **VER**, and **CREATOR**
  - Clarification of deprecated keyword code **T** with respect to results parameters stored in a BLUE file
  - Numerous updates from the NeXtMidas team
  - Release as Version 1.1.0

- 11-May-2018 (327):
  - Edits to make plain text version of ICD more readable

- Capitalized the “T” in all references to *Type xxxx* files
- Added information about Type 1998 files
- Removed references to CAM, CCD, etc., and updated where applicable to MAINSTRIKE
- Added units of measurement codes 64-76 requested via asset DRs and other avenues
- Release as Version 1.1.1